O'REILLY®

2nd Edition



Flask Web Development

DEVELOPING WEB APPLICATIONS WITH PYTHON

Miguel Grinberg

Flask Web Development

Take full creative control of your web applications with Flask, the Python-based microframework. With the second edition of this hands-on book, you'll learn Flask from the ground up by developing a complete, real-world application created by author Miguel Grinberg. This refreshed edition accounts for important technology changes that have occurred in the past three years.

Explore the framework's core functionality, and learn how to extend applications with advanced web techniques such as database migrations and an application programming interface. The first part of each chapter provides you with reference and background for the topic in question, while the second part guides you through a hands-on implementation.

If you have Python experience, you're ready to take advantage of the creative freedom Flask provides. Three sections include:

- **A thorough introduction to Flask:** explore web application development basics with Flask and an application structure appropriate for medium and large applications
- **Building Flasky:** learn how to build an open source blogging application step-by-step by reusing templates, paginating item lists, and working with rich text
- **Going the last mile:** dive into unit testing strategies, performance analysis techniques, and deployment options for your Flask application

"The second edition upholds the strong tradition of Miguel's blog posts and the book's first edition that, together, fueled my learnings about Flask, including database interactions and deployments."

—Jason Myers
author, *Essential SQLAlchemy*, 2nd Edition (O'Reilly)

Miguel Grinberg has over 25 years of experience as a software engineer. He blogs at *https://blog.miguelgrinberg.com* about a variety of topics including web development, Python, robotics, photography, and the occasional movie review.

US $44.99 CAN $59.99
ISBN: 978-1-491-99173-2

SECOND EDITION

Flask Web Development

Developing Web Applications with Python

Miguel Grinberg

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Flask Web Development

by Miguel Grinberg

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com/safari*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editor: Allyson MacDonald
Production Editor: Colleen Cole
Copyeditor: Dwight Ramsey
Proofreader: Rachel Head
Indexer: Ellen Troutman
Interior Designer: David Futato
Cover Designer: Randy Comer
Illustrator: Rebecca Demarest

March 2018: Second Edition

Revision History for the Second Edition

2018-03-02: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781491991732* for release details.

978-1-491-99173-2

[LSI]

For Alicia.

Table of Contents

Part II. Example: A Social Blogging Application

Part III. The Last Mile

Preface

Flask stands out from other frameworks because it lets developers take the driver's seat and have full creative control of their applications. Maybe you have heard the phrase "fighting the framework" before. This happens with most frameworks when you decide to solve a problem with a solution that isn't the official one. It could be that you want to use a different database engine, or maybe a different method of authenticating users. Deviating from the path set by the framework's developers will give you lots of headaches.

Flask is not like that. Do you like relational databases? Great. Flask supports them all. Maybe you prefer a NoSQL database? No problem at all. Flask works with them too. Want to use your own homegrown database engine? Don't need a database at all? Still fine. With Flask you can choose the components of your application, or even write your own if that's what you want. No questions asked!

The key to this freedom is that Flask was designed from the start to be extended. It comes with a robust core that includes the basic functionality that all web applications need and expects the rest to be provided by some of the many third-party extensions in the ecosystem—and, of course, by you.

In this book I present my workflow for developing web applications with Flask. I don't claim this to be the only true way to build applications with this framework. You should take my choices as recommendations and not as gospel.

Most software development books provide small and focused code examples that demonstrate the different features of the target technology in isolation, leaving the "glue" code that is necessary to transform these different features into a fully working application to be figured out by the reader. I take a completely different approach. All the examples I present are part of a single application that starts out very simple and is expanded in each successive chapter. This application begins life with just a few lines of code and ends as a nicely featured blogging and social networking application.

Who This Book Is For

You should have some level of Python coding experience to make the most of this book. Although the book assumes no previous Flask knowledge, Python concepts such as packages, modules, functions, decorators, and object-oriented programming are assumed to be well understood. Some familiarity with exceptions and diagnosing issues from stack traces will be very useful.

While working through the examples in this book, you will spend a great deal of time in the command line. You should feel comfortable using the command line of your operating system.

Modern web applications cannot avoid the use of HTML, CSS, and JavaScript. The example application that is developed throughout the book obviously makes use of these, but the book itself does not go into a lot of detail regarding these technologies and how they are used. Some degree of familiarity with these languages is recommended if you intend to develop complete applications without the help of a developer versed in client-side techniques.

I released the companion application to this book as open source on GitHub. Although GitHub makes it possible to download applications as regular ZIP or TAR files, I strongly recommend that you install a Git client and familiarize yourself with source code version control (at least with the basic commands to clone and check out the different versions of the application directly from the repository). The short list of commands that you'll need is shown in "How to Work with the Example Code" on page xiii. You will want to use version control for your own projects as well, so use this book as an excuse to learn Git!

Finally, this book is not a complete and exhaustive reference on the Flask framework. Most features are covered, but you should complement this book with the official Flask documentation.

How This Book Is Organized

This book is divided into three parts.

Part I, *Introduction to Flask*, explores the basics of web application development with the Flask framework and some of its extensions:

- Chapter 1 describes the installation and setup of the Flask framework.
- Chapter 2 dives straight into Flask with a basic application.
- Chapter 3 introduces the use of templates in Flask applications.
- Chapter 4 introduces web forms.
- Chapter 5 introduces databases.

- Chapter 6 introduces email support.
- Chapter 7 presents an application structure that is appropriate for medium and large applications.

Part II, *Example: A Social Blogging Application*, builds Flasky, the open source blogging and social networking application that I developed for this book:

- Chapter 8 implements a user authentication system.
- Chapter 9 implements user roles and permissions.
- Chapter 10 implements user profile pages.
- Chapter 11 creates the blogging interface.
- Chapter 12 implements followers.
- Chapter 13 implements user comments for blog posts.
- Chapter 14 implements an application programming interface (API).

Part III, *The Last Mile*, describes some important tasks not directly related to application coding that need to be considered before publishing an application:

- Chapter 15 describes different unit testing strategies in detail.
- Chapter 16 gives an overview of performance analysis techniques.
- Chapter 17 describes deployment options for Flask applications, including traditional, cloud-based, and container-based solutions.
- Chapter 18 lists additional resources.

How to Work with the Example Code

The code examples presented in this book are available for download at *https://github.com/miguelgrinberg/flasky*.

The commit history in this repository was carefully created to match the order in which concepts are presented in the book. The recommended way to work with the code is to check out the commits starting from the oldest, then move forward through the commit list as you make progress with the book. As an alternative, GitHub will also let you download each commit as a ZIP or TAR file.

If you decide to use Git to work with the source code, then you need to install the Git client, which you can download from *http://git-scm.com*. The following command downloads the example code using Git:

```
$ git clone https://github.com/miguelgrinberg/flasky.git
```

The `git clone` command installs the source code from GitHub into a *flasky2* folder that is created in the current directory. This folder does not contain just source code; a copy of the Git repository with the entire history of changes made to the application is also included.

In the first chapter you will be asked to *check out* the initial release of the application, and then, at the proper places, you will be instructed to move forward in the history. The Git command that lets you move through the change history is `git checkout`. Here is an example:

```
$ git checkout 1a
```

The `1a` referenced in the command is a *tag*: a named point in the commit history of the project. This repository is tagged according to the chapters of the book, so the `1a` tag used in the example sets the application files to the initial version used in Chapter 1. Most chapters have more than one tag associated with them, so, for example, tags `5a`, `5b`, and so on are incremental versions presented in Chapter 5.

When you run a `git checkout` command as just shown, Git will display a warning message that informs you that you are in a "detached HEAD" state. This means that you are not in any specific code branch that can accept new commits, but instead are looking at a particular commit in the middle of the change history of the project. There is no reason to be alarmed by this message, but you should keep in mind that if you make modifications to any files while in this state, issuing another `git checkout` is going to fail, because Git will not know what to do with the changes you've made. So, to be able to continue working with the project you will need to revert the files that you changed back to their original state. The easiest way to do this is with the `git reset` command:

```
$ git reset --hard
```

This command will destroy any local changes you have made, so you should save anything you don't want to lose before you use this command.

As well as checking out the source files for a version of the application, at certain points you will need to perform additional setup tasks. For example, in some cases you will need to install new Python packages, or apply updates to the database. You will be told when these are necessary.

From time to time, you may want to refresh your local repository from the one on GitHub, where bug fixes and improvements may have been applied. The commands that achieve this are:

```
$ git fetch --all
$ git fetch --tags
$ git reset --hard origin/master
```

The `git fetch` commands are used to update the commit history and the tags in your local repository from the remote one on GitHub, but none of this affects the actual source files, which are updated with the `git reset` command that follows. Once again, be aware that any time `git reset` is used you will lose any local changes you have made.

Another useful operation is to view all the differences between two versions of the application. This can be very useful to understand a change in detail. From the command line, the `git diff` command can do this. For example, to see the difference between revisions `2a` and `2b`, use:

```
$ git diff 2a 2b
```

The differences are shown as a *patch*, which is not a very intuitive format to review changes if you are not used to working with patch files. You may find that the graphical comparisons shown by GitHub are much easier to read. For example, the differences between revisions `2a` and `2b` can be viewed on GitHub at *https://github.com/miguelgrinberg/flasky/compare/2a...2b*.

Using Code Examples

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Flask Web Development*, 2nd Edition, by Miguel Grinberg (O'Reilly). Copyright 2018 Miguel Grinberg, 978-1-491-99173-2."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for command-line output and program listings, as well as within paragraphs to refer to commands and to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic` or angle brackets (<>)
: Indicates text that should be replaced with user-supplied values or by values determined by context.

This element signifies a tip or suggestion.

This element signifies a general note.

This element indicates a warning or caution.

O'Reilly Safari

Safari (formerly Safari Books Online) is a membership-based training and reference platform for enterprise, government, educators, and individuals.

Members have access to thousands of books, training videos, Learning Paths, interactive tutorials, and curated playlists from over 250 publishers, including O'Reilly Media, Harvard Business Review, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Adobe, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, and Course Technology, among others.

For more information, please visit *http://oreilly.com/safari*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *http://bit.ly/flask-web-dev2*.

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

I could not have written this book alone. I have received a lot of help from family, coworkers, old friends, and new friends I've made along the way.

I'd like to thank Brendan Kohler for his detailed technical review and for his help in giving shape to the chapter on application programming interfaces. I'm also in debt to David Baumgold, Todd Brunhoff, Cecil Rock, and Matthew Hugues, who reviewed the manuscript at different stages of completion and gave me very useful advice regarding what to cover and how to organize the material.

Writing the code examples for this book was a considerable effort. I appreciate the help of Daniel Hofmann, who did a thorough code review of the application and pointed out several improvements. I'm also thankful to my teenage son, Dylan Grinberg, who suspended his Minecraft addiction for a few weekends and helped me test the code under several platforms.

O'Reilly has a wonderful program called Early Release that allows impatient readers to have access to books while they are being written. Some of my Early Release read-

ers went the extra mile and engaged in useful conversations regarding their experience working through the book, leading to significant improvements. I'd like to acknowledge Sundeep Gupta, Dan Caron, Brian Wisti, and Cody Scott in particular for the contributions they've made to this book.

The staff at O'Reilly Media have always been there for me. Above all I'd like to recognize my wonderful editor, Meghan Blanchette, for her support, advice, and assistance from the very first day we met. Meg made the experience of writing my first book a memorable one.

To conclude, I would like to give a big thank you to the awesome Flask community.

Additional Thanks for the Second Edition

I'd like to thank Ally MacDonald, my editor for the second edition of this book, and also Susan Conant, Rachel Roumeliotis, and the whole team at O'Reilly Media for their continued support.

The technical reviewers for this edition did a wonderful job pointing out areas to improve and providing me with new perspectives. I'd like to recognize Lorena Mesa, Diane Chen, and Jesse Smith for their great contributions through their feedback and suggestions. I also greatly appreciate the help of my son, Dylan Grinberg, who painstakingly tested all the code examples.

PART I

Introduction to Flask

CHAPTER 1

Installation

Flask is a small framework by most standards—small enough to be called a "micro-framework," and small enough that once you become familiar with it, you will likely be able to read and understand all of its source code.

But being small does not mean that it does less than other frameworks. Flask was designed as an extensible framework from the ground up; it provides a solid core with the basic services, while *extensions* provide the rest. Because you can pick and choose the extension packages that you want, you end up with a lean stack that has no bloat and does exactly what you need.

Flask has three main dependencies. The routing, debugging, and Web Server Gateway Interface (WSGI) subsystems come from Werkzeug; the template support is provided by Jinja2; and the command-line integration comes from Click. These dependencies are all authored by Armin Ronacher, the author of Flask.

Flask has no native support for accessing databases, validating web forms, authenticating users, or other high-level tasks. These and many other key services most web applications need are available through extensions that integrate with the core packages. As a developer, you have the power to cherry-pick the extensions that work best for your project, or even write your own if you feel inclined to. This is in contrast with a larger framework, where most choices have been made for you and are hard or sometimes impossible to change.

In this chapter, you will learn how to install Flask. The only requirement is a computer with Python installed.

The code examples in this book have been verified to work with Python 3.5 and 3.6. Python 2.7 can also be used if desired, but given that this version of Python is not going to be maintained after the year 2020, it is strongly recommended that you use the 3.x versions.

If you plan to use a Microsoft Windows computer to work with the example code, you have to decide if you want to use a "native" approach based on Windows tools, or set up your computer in a way that allows you to adopt the more mainstream Unix-based toolset. The code presented in this book is largely compatible with both approaches. In the few cases where the approaches differ, the Unix solution is followed, and alternatives for Windows are noted.

If you decide to follow a Unix workflow, you have a few options. If you are using Windows 10, you can enable the Windows Subsystem for Linux (WSL), which is an officially supported feature that creates an Ubuntu Linux installation that runs alongside the native Windows interface, giving you access to a *bash* shell and the complete set of Unix-based tools. If WSL is not available on your system, another good option is Cygwin, an open-source project that emulates the POSIX subsystem used by Unix and provides ports of a large number of Unix tools.

Creating the Application Directory

To begin, you need to create the directory that will host the example code, which is available in a GitHub repository. As discussed in "How to Work with the Example Code" on page xiii, the most convenient way to do this is by checking out the code directly from GitHub using a Git client. The following commands download the example code from GitHub and initialize the application to version `1a`, which is the initial version you will work with:

```
$ git clone https://github.com/miguelgrinberg/flasky.git
$ cd flasky
$ git checkout 1a
```

If you prefer not to use Git and instead manually type or copy the code, you can simply create an empty application directory as follows:

```
$ mkdir flasky
$ cd flasky
```

Virtual Environments

Now that you have the application directory created, it is time to install Flask. The most convenient way to do that is to use a *virtual environment*. A virtual environment

is a copy of the Python interpreter into which you can install packages privately, without affecting the global Python interpreter installed in your system.

Virtual environments are very useful because they prevent package clutter and version conflicts in the system's Python interpreter. Creating a virtual environment for each project ensures that applications have access only to the packages that they use, while the global interpreter remains neat and clean and serves only as a source from which more virtual environments can be created. As an added benefit, unlike the system-wide Python interpreter, virtual environments can be created and managed without administrator rights.

Creating a Virtual Environment with Python 3

The creation of virtual environments is an area where Python 3 and Python 2 interpreters differ. With Python 3, virtual environments are supported natively by the `venv` package that is part of the Python standard library.

> If you are using the stock Python 3 interpreter on an Ubuntu Linux system, the standard `venv` package is not installed by default. To add it to your system, install the `python3-venv` package as follows:
>
> ```
> $ sudo apt-get install python3-venv
> ```

The command that creates a virtual environment has the following structure:

```
$ python3 -m venv virtual-environment-name
```

The `-m venv` option runs the `venv` package from the standard library as a standalone script, passing the desired name as an argument.

You are now going to create a virtual environment inside the *flasky* directory. A commonly used convention for virtual environments is to call them *venv*, but you can use a different name if you prefer. Make sure your current directory is set to *flasky*, and then run this command:

```
$ python3 -m venv venv
```

After the command completes, you will have a subdirectory with the name *venv* inside *flasky*, with a brand-new virtual environment that contains a Python interpreter for exclusive use by this project.

Creating a Virtual Environment with Python 2

Python 2 does not have a `venv` package. In this version of the Python interpreter, virtual environments are created with the third-party utility *virtualenv*.

Make sure your current directory is set to *flasky*, and then use one of the following two commands, depending on your operating system. If you are using Linux or macOS, the command is:

```
$ sudo pip install virtualenv
```

If you are using Microsoft Windows, make sure you open a command prompt window using the "Run as Administrator" option, and then run this command:

```
$ pip install virtualenv
```

The `virtualenv` command takes the name of the virtual environment as its argument. Make sure your current directory is set to *flasky*, and then run the following command to create a virtual environment called *venv*:

```
$ virtualenv venv
New python executable in venv/bin/python2.7
Also creating executable in venv/bin/python
Installing setuptools, pip, wheel...done.
```

A subdirectory with the *venv* name will be created in the current directory, and all files associated with the virtual environment will be inside it.

Working with a Virtual Environment

When you want to start using a virtual environment, you have to "activate" it. If you are using a Linux or macOS computer, you can activate the virtual environment with this command:

```
$ source venv/bin/activate
```

If you are using Microsoft Windows, the activation command is:

```
$ venv\Scripts\activate
```

When a virtual environment is activated, the location of its Python interpreter is added to the `PATH` environment variable in your current command session, which determines where to look for executable files. To remind you that you have activated a virtual environment, the activation command modifies your command prompt to include the name of the environment:

```
(venv) $
```

After a virtual environment is activated, typing `python` at the command prompt will invoke the interpreter from the virtual environment instead of the system-wide interpreter. If you are using more than one command prompt window, you have to activate the virtual environment in each of them.

> While activating a virtual environment is usually the most convenient option, you can also use a virtual environment without activating it. For example, you can start a Python console for the *venv* virtual environment by running `venv/bin/python` on Linux or macOS, or `venv\Scripts\python` on Microsoft Windows.

When you are done working with the virtual environment, type **`deactivate`** at the command prompt to restore the `PATH` environment variable for your terminal session and the command prompt to their original states.

Installing Python Packages with pip

Python packages are installed with the *pip* package manager, which is included in all virtual environments. Like the `python` command, typing `pip` in a command prompt session will invoke the version of this tool that belongs to the activated virtual environment.

To install Flask into the virtual environment, make sure the *venv* virtual environment is activated, and then run the following command:

```
(venv) $ pip install flask
```

When you execute this command, *pip* will not only install Flask, but also all of its dependencies. You can check what packages are installed in the virtual environment at any time using the `pip freeze` command:

```
(venv) $ pip freeze
click==6.7
Flask==0.12.2
itsdangerous==0.24
Jinja2==2.9.6
MarkupSafe==1.0
Werkzeug==0.12.2
```

The output of `pip freeze` includes detailed version numbers for each installed package. The version numbers that you get are likely going to be different from the ones shown here.

You can also verify that Flask was correctly installed by starting the Python interpreter and trying to import it:

```
(venv) $ python
>>> import flask
>>>
```

If no errors appear, you can congratulate yourself: you are ready for the next chapter, where you will write your first web application.

CHAPTER 2

Basic Application Structure

In this chapter, you will learn about the different parts of a Flask application. You will also write and run your first Flask web application.

Initialization

All Flask applications must create an *application instance*. The web server passes all requests it receives from clients to this object for handling, using a protocol called Web Server Gateway Interface (WSGI, pronounced "wiz-ghee"). The application instance is an object of class `Flask`, usually created as follows:

```
from flask import Flask
app = Flask(__name__)
```

The only required argument to the `Flask` class constructor is the name of the main module or package of the application. For most applications, Python's `__name__` variable is the correct value for this argument.

> The `__name__` argument that is passed to the Flask application constructor is a source of confusion among new Flask developers. Flask uses this argument to determine the location of the application, which in turn allows it to locate other files that are part of the application, such as images and templates.

Later you will learn more complex ways to initialize an application, but for simple applications this is all that is needed.

Routes and View Functions

Clients such as web browsers send *requests* to the web server, which in turn sends them to the Flask application instance. The Flask application instance needs to know what code it needs to run for each URL requested, so it keeps a mapping of URLs to Python functions. The association between a URL and the function that handles it is called a *route*.

The most convenient way to define a route in a Flask application is through the `app.route` decorator exposed by the application instance. The following example shows how a route is declared using this decorator:

```
@app.route('/')
def index():
    return '<h1>Hello World!</h1>'
```

Decorators are a standard feature of the Python language. A common use of decorators is to register functions as handler functions to be invoked when certain events occur.

The previous example registers function `index()` as the handler for the application's root URL. While the `app.route` decorator is the preferred method to register view functions, Flask also offers a more traditional way to set up the application routes with the `app.add_url_rule()` method, which in its most basic form takes three arguments: the URL, the endpoint name, and the view function. The following example uses `app.add_url_rule()` to register an `index()` function that is equivalent to the one shown previously:

```
def index():
    return '<h1>Hello World!</h1>'

app.add_url_rule('/', 'index', index)
```

Functions like `index()` that handle application URLs are called *view functions*. If the application is deployed on a server associated with the *www.example.com* domain name, then navigating to *http://www.example.com/* in your browser would trigger `index()` to run on the server. The return value of this view function is the *response* the client receives. If the client is a web browser, this response is the document that is displayed to the user in the browser window. A response returned by a view function can be a simple string with HTML content, but it can also take more complex forms, as you will see later.

> Embedding response strings with HTML code in Python source files leads to code that is difficult to maintain. The examples in this chapter do it only to introduce the concept of responses. You will learn a better way to generate HTML responses in Chapter 3.

If you pay attention to how some URLs for services that you use every day are formed, you will notice that many have variable sections. For example, the URL for your Facebook profile page has the format *https://www.facebook.com/<your-name>*, which includes your username, making it different for each user. Flask supports these types of URLs using a special syntax in the `app.route` decorator. The following example defines a route that has a dynamic component:

```
@app.route('/user/<name>')
def user(name):
    return '<h1>Hello, {}!</h1>'.format(name)
```

The portion of the route URL enclosed in angle brackets is the dynamic part. Any URLs that match the static portions will be mapped to this route, and when the view function is invoked, the dynamic component will be passed as an argument. In the preceding example, the `name` argument is used to generate a response that includes a personalized greeting.

The dynamic components in routes are strings by default but can also be of different types. For example, the route `/user/<int:id>` would match only URLs that have an integer in the `id` dynamic segment, such as */user/123*. Flask supports the types `string`, `int`, `float`, and `path` for routes. The `path` type is a special string type that can include forward slashes, unlike the `string` type.

A Complete Application

In the previous sections you learned about the different parts of a Flask web application, and now it is time to write your first one. The *hello.py* application script shown in Example 2-1 defines an application instance and a single route and view function, as described earlier.

Example 2-1. hello.py: A complete Flask application

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def index():
    return '<h1>Hello World!</h1>'
```

If you have cloned the application's Git repository on GitHub, you can now run `git checkout 2a` to check out this version of the application.

Development Web Server

Flask applications include a development web server that can be started with the `flask run` command. This command looks for the name of the Python script that contains the application instance in the `FLASK_APP` environment variable.

To start the *hello.py* application from the previous section, first make sure the virtual environment you created earlier is activated and has Flask installed in it. For Linux and macOS users, start the web server as follows:

```
(venv) $ export FLASK_APP=hello.py
(venv) $ flask run
 * Serving Flask app "hello"
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

For Microsoft Windows users, the only difference is in how the `FLASK_APP` environment variable is set:

```
(venv) $ set FLASK_APP=hello.py
(venv) $ flask run
 * Serving Flask app "hello"
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

Once the server starts up, it goes into a loop that accepts requests and services them. This loop continues until you stop the application by pressing Ctrl+C.

With the server running, open your web browser and type **`http://localhost:5000/`** in the address bar. Figure 2-1 shows what you'll see after connecting to the application.

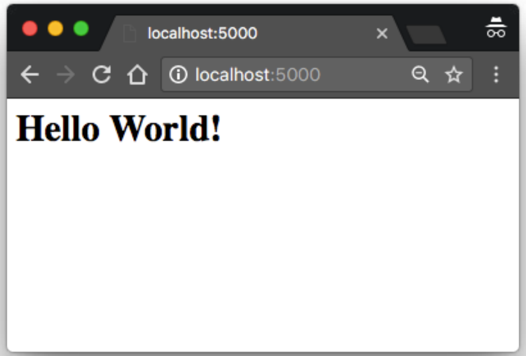


Figure 2-1. hello.py Flask application

If you type anything else after the base URL, the application will not know how to handle it and will return an error code 404 to the browser—the familiar error that you get when you navigate to a web page that does not exist.

> The web server provided by Flask is intended to be used only for development and testing. You will learn about production web servers in Chapter 17.

> The Flask development web server can also be started programmatically by invoking the `app.run()` method. Older versions of Flask that did not have the `flask` command required the server to be started by running the application's main script, which had to include the following snippet at the end:
>
> ```
> if __name__ == '__main__':
>     app.run()
> ```
>
> While the `flask run` command makes this practice unnecessary, the `app.run()` method can still be useful on certain occasions, such as unit testing, as you will learn in Chapter 15.

Dynamic Routes

The second version of the application, shown in Example 2-2, adds a second route that is dynamic. When you visit the dynamic URL in your browser, you are presented with a personalized greeting that includes the name provided in the URL.

Example 2-2. hello.py: Flask application with a dynamic route

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def index():
    return '<h1>Hello World!</h1>'

@app.route('/user/<name>')
def user(name):
    return '<h1>Hello, {}!</h1>'.format(name)
```

> If you have cloned the application's Git repository on GitHub, you can now run `git checkout 2b` to check out this version of the application.

To test the dynamic route, make sure the server is running and then navigate to *http://localhost:5000/user/Dave*. The application will respond with the personalized greeting using the `name` dynamic argument. Try using different names in the URL to see how the view function always generates the response based on the name given. An example is shown in Figure 2-2.

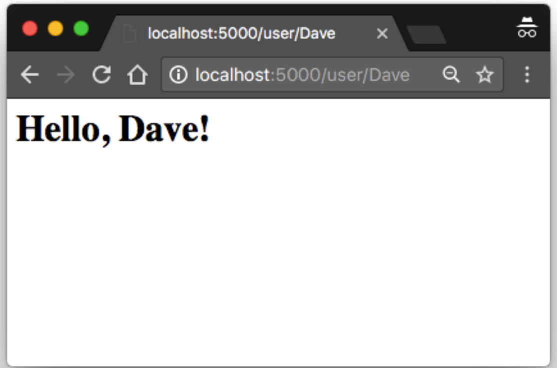


Figure 2-2. Dynamic route

Debug Mode

Flask applications can optionally be executed in *debug mode*. In this mode, two very convenient modules of the development server called the *reloader* and the *debugger* are enabled by default.

When the reloader is enabled, Flask watches all the source code files of your project and automatically restarts the server when any of the files are modified. Having a server running with the reloader enabled is extremely useful during development, because every time you modify and save a source file, the server automatically restarts and picks up the change.

The debugger is a web-based tool that appears in your browser when your application raises an unhandled exception. The web browser window transforms into an interactive stack trace that allows you to inspect source code and evaluate expressions in any place in the call stack. You can see how the debugger looks in Figure 2-3.

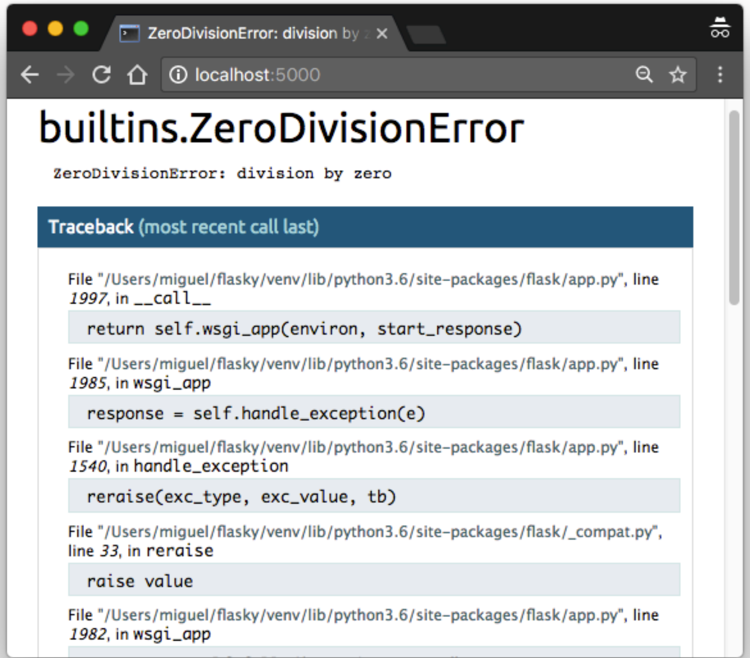


Figure 2-3. Flask debugger

By default, debug mode is disabled. To enable it, set a `FLASK_DEBUG=1` environment variable before invoking `flask run`:

```
(venv) $ export FLASK_APP=hello.py
(venv) $ export FLASK_DEBUG=1
(venv) $ flask run
 * Serving Flask app "hello"
 * Forcing debug mode on
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 273-181-528
```

If you are using Microsoft Windows, use `set` instead of `export` to set the environment variables.

If you start your server with the `app.run()` method, the `FLASK_APP` and `FLASK_DEBUG` environment variables are not used. To enable debug mode programmatically, use `app.run(debug=True)`.

Never enable debug mode on a production server. The debugger in particular allows the client to request remote code execution, so it makes your production server vulnerable to attacks. As a simple protection measure, the debugger needs to be activated with a PIN, printed to the console by the `flask run` command.

Command-Line Options

The `flask` command supports a number of options. To see what's available, you can run `flask --help` or just `flask` without any arguments:

```
(venv) $ flask --help
Usage: flask [OPTIONS] COMMAND [ARGS]...

  This shell command acts as general utility script for Flask applications.

  It loads the application configured (through the FLASK_APP environment
  variable) and then provides commands either provided by the application or
  Flask itself.

  The most useful commands are the "run" and "shell" command.

  Example usage:

    $ export FLASK_APP=hello.py
    $ export FLASK_DEBUG=1
    $ flask run

Options:
  --version  Show the flask version
  --help     Show this message and exit.

Commands:
  run     Runs a development server.
  shell   Runs a shell in the app context.
```

The `flask shell` command is used to start a Python shell session in the context of the application. You can use this session to run maintenance tasks or tests, or to debug issues. Actual examples where this command is useful will be presented later, in several chapters.

You are already familiar with the `flask run` command, which, as its name implies, runs the application with the development web server. This command has many options:

```
(venv) $ flask run --help
Usage: flask run [OPTIONS]

  Runs a local development server for the Flask application.

  This local server is recommended for development purposes only but it can
  also be used for simple intranet deployments.  By default it will not
  support any sort of concurrency at all to simplify debugging.  This can be
  changed with the --with-threads option which will enable basic
  multithreading.

  The reloader and debugger are by default enabled if the debug flag of
  Flask is enabled and disabled otherwise.

Options:
  -h, --host TEXT                 The interface to bind to.
  -p, --port INTEGER              The port to bind to.
  --reload / --no-reload          Enable or disable the reloader.  By default
                                  the reloader is active if debug is enabled.
  --debugger / --no-debugger      Enable or disable the debugger.  By default
                                  the debugger is active if debug is enabled.
  --eager-loading / --lazy-loader
                                  Enable or disable eager loading.  By default
                                  eager loading is enabled if the reloader is
                                  disabled.
  --with-threads / --without-threads
                                  Enable or disable multithreading.
  --help                          Show this message and exit.
```

The `--host` argument is particularly useful because it tells the web server what network interface to listen to for connections from clients. By default, Flask's development web server listens for connections on *localhost*, so only connections originating from the computer running the server are accepted. The following command makes the web server listen for connections on the public network interface, enabling other computers in the same network to connect as well:

```
(venv) $ flask run --host 0.0.0.0
 * Serving Flask app "hello"
 * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
```

The web server should now be accessible from any computer in the network at *http://a.b.c.d:5000*, where *a.b.c.d* is the IP address of the computer running the server in your network.

The `--reload`, `--no-reload`, `--debugger`, and `--no-debugger` options provide a greater degree of control on top of the debug mode setting. For example, if debug

mode is enabled, `--no-debugger` can be used to turn off the debugger, while keeping debug mode and the reloader enabled.

The Request-Response Cycle

Now that you have played with a basic Flask application, you might want to know more about how Flask works its magic. The following sections describe some of the design aspects of the framework.

Application and Request Contexts

When Flask receives a request from a client, it needs to make a few objects available to the view function that will handle it. A good example is the *request object*, which encapsulates the HTTP request sent by the client.

The obvious way in which Flask could give a view function access to the request object is by sending it as an argument, but that would require every single view function in the application to have an extra argument. Things get more complicated if you consider that the request object is not the only object that view functions might need to access to fulfill a request.

To avoid cluttering view functions with lots of arguments that may not always be needed, Flask uses *contexts* to temporarily make certain objects globally accessible. Thanks to contexts, view functions like the following one can be written:

```
from flask import request

@app.route('/')
def index():
    user_agent = request.headers.get('User-Agent')
    return '<p>Your browser is {}</p>'.format(user_agent)
```

Note how in this view function, `request` is used as if it were a global variable. In reality, `request` cannot be a global variable; in a multithreaded server several threads can be working on different requests from different clients all at the same time, so each thread needs to see a different object in `request`. Contexts enable Flask to make certain variables globally accessible to a thread without interfering with the other threads.

> A thread is the smallest sequence of instructions that can be managed independently. It is common for a process to have multiple active threads, sometimes sharing resources such as memory or file handles. Multithreaded web servers start a pool of threads and select a thread from the pool to handle each incoming request.

There are two contexts in Flask: the *application context* and the *request context*. Table 2-1 shows the variables exposed by each of these contexts.

Table 2-1. Flask context globals

Variable name	Context	Description
`current_app`	Application context	The application instance for the active application.
`g`	Application context	An object that the application can use for temporary storage during the handling of a request. This variable is reset with each request.
`request`	Request context	The request object, which encapsulates the contents of an HTTP request sent by the client.
`session`	Request context	The user session, a dictionary that the application can use to store values that are "remembered" between requests.

Flask activates (or *pushes*) the application and request contexts before dispatching a request to the application, and removes them after the request is handled. When the application context is pushed, the `current_app` and `g` variables become available to the thread. Likewise, when the request context is pushed, `request` and `session` become available as well. If any of these variables are accessed without an active application or request context, an error is generated. The four context variables will be covered in detail in this and later chapters, so don't worry if you don't understand why they are useful yet.

The following Python shell session demonstrates how the application context works:

```
>>> from hello import app
>>> from flask import current_app
>>> current_app.name
Traceback (most recent call last):
...
RuntimeError: working outside of application context
>>> app_ctx = app.app_context()
>>> app_ctx.push()
>>> current_app.name
'hello'
>>> app_ctx.pop()
```

In this example, `current_app.name` fails when there is no application context active but becomes valid once an application context for the application is pushed. Note how an application context is obtained by invoking `app.app_context()` on the application instance.

Request Dispatching

When the application receives a request from a client, it needs to find out what view function to invoke to service it. For this task, Flask looks up the URL given in the

request in the application's *URL map*, which contains a mapping of URLs to the view functions that handle them. Flask builds this map using the data provided in the `app.route` decorator, or the equivalent non-decorator version, `app.add_url_rule()`.

To see what the URL map in a Flask application looks like, you can inspect the map created for *hello.py* in the Python shell. Before you try this, make sure that your virtual environment is activated:

```
(venv) $ python
>>> from hello import app
>>> app.url_map
Map([<Rule '/' (HEAD, OPTIONS, GET) -> index>,
 <Rule '/static/<filename>' (HEAD, OPTIONS, GET) -> static>,
 <Rule '/user/<name>' (HEAD, OPTIONS, GET) -> user>])
```

The `/` and `/user/<name>` routes were defined by the `app.route` decorators in the application. The `/static/<filename>` route is a special route added by Flask to give access to static files. You will learn more about static files in Chapter 3.

The `(HEAD, OPTIONS, GET)` elements shown in the URL map are the *request methods* that are handled by the routes. The HTTP specification defines that all requests are issued with a method, which normally indicates what action the client is asking the server to perform. Flask attaches methods to each route so that different request methods sent to the same URL can be handled by different view functions. The `HEAD` and `OPTIONS` methods are managed automatically by Flask, so in practice it can be said that in this application the three routes in the URL map are attached to the `GET` method, which is used when the client wants to request information such as a web page. You will learn how to create routes for other request methods in Chapter 4.

The Request Object

You have seen that Flask exposes a request object as a context variable named `request`. This is an extremely useful object that contains all the information that the client included in the HTTP request. Table 2-2 enumerates the most commonly used attributes and methods of the Flask request object.

Table 2-2. Flask request object

Attribute or Method	Description
`form`	A dictionary with all the form fields submitted with the request.
`args`	A dictionary with all the arguments passed in the query string of the URL.
`values`	A dictionary that combines the values in `form` and `args`.
`cookies`	A dictionary with all the cookies included in the request.
`headers`	A dictionary with all the HTTP headers included in the request.
`files`	A dictionary with all the file uploads included with the request.

Attribute or Method	Description
get_data()	Returns the buffered data from the request body.
get_json()	Returns a Python dictionary with the parsed JSON included in the body of the request.
blueprint	The name of the Flask blueprint that is handling the request. Blueprints are introduced in Chapter 7.
endpoint	The name of the Flask endpoint that is handling the request. Flask uses the name of the view function as the endpoint name for a route.
method	The HTTP request method, such as GET or POST.
scheme	The URL scheme (http or https).
is_secure()	Returns True if the request came through a secure (HTTPS) connection.
host	The host defined in the request, including the port number if given by the client.
path	The path portion of the URL.
query_string	The query string portion of the URL, as a raw binary value.
full_path	The path and query string portions of the URL.
url	The complete URL requested by the client.
base_url	Same as url, but without the query string component.
remote_addr	The IP address of the client.
environ	The raw WSGI environment dictionary for the request.

Request Hooks

Sometimes it is useful to execute code before or after each request is processed. For example, at the start of each request it may be necessary to create a database connection or authenticate the user making the request. Instead of duplicating the code that performs these actions in every view function, Flask gives you the option to register common functions to be invoked before or after a request is dispatched.

Request hooks are implemented as decorators. These are the four hooks supported by Flask:

`before_request`
: Registers a function to run before each request.

`before_first_request`
: Registers a function to run only before the first request is handled. This can be a convenient way to add server initialization tasks.

`after_request`
: Registers a function to run after each request, but only if no unhandled exceptions occurred.

`teardown_request`
: Registers a function to run after each request, even if unhandled exceptions occurred.

A common pattern to share data between request hook functions and view functions is to use the `g` context global as storage. For example, a `before_request` handler can load the logged-in user from the database and store it in `g.user`. Later, when the view function is invoked, it can retrieve the user from there.

Examples of request hooks will be shown in future chapters, so don't worry if the purpose of these hooks does not quite make sense yet.

Responses

When Flask invokes a view function, it expects its return value to be the response to the request. In most cases the response is a simple string that is sent back to the client as an HTML page.

But the HTTP protocol requires more than a string as a response to a request. A very important part of the HTTP response is the *status code*, which Flask by default sets to 200, the code that indicates that the request was carried out successfully.

When a view function needs to respond with a different status code, it can add the numeric code as a second return value after the response text. For example, the following view function returns a 400 status code, the code for a bad request error:

```
@app.route('/')
def index():
    return '<h1>Bad Request</h1>', 400
```

Responses returned by view functions can also take a third argument, a dictionary of headers that are added to the HTTP response. You will see an example of custom response headers in Chapter 14.

Instead of returning one, two, or three values as a tuple, Flask view functions have the option of returning a *response object*. The `make_response()` function takes one, two, or three arguments, the same values that can be returned from a view function, and returns an equivalent response object. Sometimes it is useful to generate the response object inside the view function, and then use its methods to further configure the response. The following example creates a response object and then sets a cookie in it:

```
from flask import make_response

@app.route('/')
def index():
    response = make_response('<h1>This document carries a cookie!</h1>')
    response.set_cookie('answer', '42')
    return response
```

Table 2-3 shows the most commonly used attributes and methods available in response objects.

Table 2-3. Flask response object

Attribute or Method	Description
`status_code`	The numeric HTTP status code
`headers`	A dictionary-like object with all the headers that will be sent with the response
`set_cookie()`	Adds a cookie to the response
`delete_cookie()`	Removes a cookie
`content_length`	The length of the response body
`content_type`	The media type of the response body
`set_data()`	Sets the response body as a string or bytes value
`get_data()`	Gets the response body

There is a special type of response called a *redirect*. This response does not include a page document; it just gives the browser a new URL to navigate to. A very common use of redirects is when working with web forms, as you will learn in Chapter 4.

A redirect is typically indicated with a 302 response status code and the URL to go to given in a `Location` header. A redirect response can be generated manually with a three-value return or with a response object, but given its frequent use, Flask provides a `redirect()` helper function that creates this type of response:

```
from flask import redirect

@app.route('/')
def index():
    return redirect('http://www.example.com')
```

Another special response is issued with the `abort()` function, which is used for error handling. The following example returns status code 404 if the `id` dynamic argument given in the URL does not represent a valid user:

```
from flask import abort

@app.route('/user/<id>')
def get_user(id):
    user = load_user(id)
    if not user:
        abort(404)
    return '<h1>Hello, {}</h1>'.format(user.name)
```

Note that `abort()` does not return control back to the function because it raises an exception.

Flask Extensions

Flask is designed to be extended. It intentionally stays out of areas of important functionality such as database and user authentication, giving you the freedom to select the packages that fit your application the best, or to write your own if you so desire.

A great variety of Flask extensions for many different purposes have been created by the community, and if that is not enough, any standard Python package or library can be used as well. You will use your first Flask extension in Chapter 3.

This chapter introduced the concept of responses to requests, but there is a lot more to say about responses. Flask provides very good support for generating responses using *templates*, and this is such an important topic that the next chapter is dedicated to it.

CHAPTER 3
Templates

The key to writing applications that are easy to maintain is to write clean and well-structured code. The examples that you have seen so far are too simple to demonstrate this, but Flask view functions have two completely independent purposes disguised as one, which creates a problem.

The obvious task of a view function is to generate a response to a request, as you have seen in the examples shown in Chapter 2. For the simplest requests this is enough, but in many cases a request also triggers a change in the state of the application, and the view function is where this change is generated.

For example, consider a user who is registering a new account on a website. The user types an email address and a password in a web form and clicks the Submit button. On the server, a request with the data provided by the user arrives, and Flask dispatches it to the view function that handles registration requests. This view function needs to talk to the database to get the new user added, and then generate a response to send back to the browser that includes a success or failure message. These two types of tasks are formally called *business logic* and *presentation logic*, respectively.

Mixing business and presentation logic leads to code that is hard to understand and maintain. Imagine having to build the HTML code for a large table by concatenating data obtained from the database with the necessary HTML string literals. Moving the presentation logic into *templates* helps improve the maintainability of the application.

A template is a file that contains the text of a response, with placeholder variables for the dynamic parts that will be known only in the context of a request. The process that replaces the variables with actual values and returns a final response string is called *rendering*. For the task of rendering templates, Flask uses a powerful template engine called *Jinja2*.

The Jinja2 Template Engine

In its simplest form, a Jinja2 template is a file that contains the text of a response. Example 3-1 shows a Jinja2 template that matches the response of the `index()` view function of Example 2-1.

Example 3-1. templates/index.html: Jinja2 template

```
<h1>Hello World!</h1>
```

The response returned by the `user()` view function of Example 2-2 has a dynamic component, which is represented by a *variable*. Example 3-2 shows the template that implements this response.

Example 3-2. templates/user.html: Jinja2 template

```
<h1>Hello, {{ name }}!</h1>
```

Rendering Templates

By default Flask looks for templates in a *templates* subdirectory located inside the main application directory. For the next version of *hello.py*, you need to create the *templates* subdirectory and store the templates defined in the previous examples in it as *index.html* and *user.html*, respectively.

The view functions in the application need to be modified to render these templates. Example 3-3 shows these changes.

Example 3-3. hello.py: rendering a template

```
from flask import Flask, render_template

# ...

@app.route('/')
def index():
    return render_template('index.html')

@app.route('/user/<name>')
def user(name):
    return render_template('user.html', name=name)
```

The function render_template() provided by Flask integrates the Jinja2 template engine with the application. This function takes the filename of the template as its first argument. Any additional arguments are key-value pairs that represent actual values for variables referenced in the template. In this example, the second template is receiving a name variable.

Keyword arguments like name=name in the previous example are fairly common, but they may seem confusing and hard to understand if you are not used to them. The "name" on the left side represents the argument name, which is used in the placeholder written in the template. The "name" on the right side is a variable in the current scope that provides the value for the argument of the same name. While this is a common pattern, using the same variable name on both sides is not required.

> If you have cloned the application's Git repository on GitHub, you can run git checkout 3a to check out this version of the application.

Variables

The {{ name }} construct used in the template shown in Example 3-2 references a variable, a special placeholder that tells the template engine that the value that goes in that place should be obtained from data provided at the time the template is rendered.

Jinja2 recognizes variables of any type, even complex types such as lists, dictionaries, and objects. The following are some more examples of variables used in templates:

```
<p>A value from a dictionary: {{ mydict['key'] }}.</p>
<p>A value from a list: {{ mylist[3] }}.</p>
<p>A value from a list, with a variable index: {{ mylist[myintvar] }}.</p>
<p>A value from an object's method: {{ myobj.somemethod() }}.</p>
```

Variables can be modified with *filters*, which are added after the variable name with a pipe character as separator. For example, the following template shows the name variable capitalized:

```
Hello, {{ name|capitalize }}
```

Table 3-1 lists some of the commonly used filters that come with Jinja2.

Table 3-1. Jinja2 variable filters

Filter name	Description
safe	Renders the value without applying escaping
capitalize	Converts the first character of the value to uppercase and the rest to lowercase
lower	Converts the value to lowercase characters
upper	Converts the value to uppercase characters
title	Capitalizes each word in the value
trim	Removes leading and trailing whitespace from the value
striptags	Removes any HTML tags from the value before rendering

The safe filter is interesting to highlight. By default Jinja2 *escapes* all variables for security purposes. For example, if a variable is set to the value '<h1>Hello</h1>', Jinja2 will render the string as '<h1>Hello</h1>', which will cause the h1 element to be displayed and not interpreted by the browser. Many times it is necessary to display HTML code stored in variables, and for those cases the safe filter is used.

Never use the safe filter on values that aren't trusted, such as text entered by users on web forms.

The complete list of filters can be obtained from the official Jinja2 documentation.

Control Structures

Jinja2 offers several control structures that can be used to alter the flow of the template. This section introduces some of the most useful ones with simple examples.

The following example shows how conditional statements can be entered in a template:

```
{% if user %}
    Hello, {{ user }}!
{% else %}
    Hello, Stranger!
{% endif %}
```

Another common need in templates is to render a list of elements. This example shows how this can be done with a for loop:

```
<ul>
    {% for comment in comments %}
```

```
        <li>{{ comment }}</li>
    {% endfor %}
</ul>
```

Jinja2 also supports *macros*, which are similar to functions in Python code. For example:

```
{% macro render_comment(comment) %}
    <li>{{ comment }}</li>
{% endmacro %}

<ul>
    {% for comment in comments %}
        {{ render_comment(comment) }}
    {% endfor %}
</ul>
```

To make macros more reusable, they can be stored in standalone files that are then *imported* from all the templates that need them:

```
{% import 'macros.html' as macros %}
<ul>
    {% for comment in comments %}
        {{ macros.render_comment(comment) }}
    {% endfor %}
</ul>
```

Portions of template code that need to be repeated in several places can be stored in a separate file and *included* from all the templates to avoid repetition:

```
{% include 'common.html' %}
```

Yet another powerful way to reuse is through template inheritance, which is similar to class inheritance in Python code. First, a base template is created with the name *base.html*:

```
<html>
<head>
    {% block head %}
    <title>{% block title %}{% endblock %} - My Application</title>
    {% endblock %}
</head>
<body>
    {% block body %}
    {% endblock %}
</body>
</html>
```

Base templates define *blocks* that can be overridden by derived templates. The Jinja2 `block` and `endblock` directives define blocks of content that are added to the base template. In this example, there are blocks called `head`, `title`, and `body`; note that `title` is contained by `head`. The following example is a derived template of the base template:

```
{% extends "base.html" %}
{% block title %}Index{% endblock %}
{% block head %}
    {{ super() }}
    <style>
    </style>
{% endblock %}
{% block body %}
<h1>Hello, World!</h1>
{% endblock %}
```

The `extends` directive declares that this template derives from *base.html*. This directive is followed by new definitions for the three blocks defined in the base template, which are inserted in the proper places. When a block has some content in both the base and derived templates, the content from the derived template is used. Within this block, the derived template can call `super()` to reference the contents of the block in the base template. In the preceding example, this is done in the `head` block.

Real-world usage of all the control structures presented in this section will be shown later, so you will have the opportunity to see how they work.

Bootstrap Integration with Flask-Bootstrap

Bootstrap is an open-source web browser framework from Twitter that provides user interface components that help create clean and attractive web pages that are compatible with all modern web browsers used on desktop and mobile platforms.

Bootstrap is a client-side framework, so the server is not directly involved with it. All the server needs to do is provide HTML responses that reference Bootstrap's Cascading Style Sheets (CSS) and JavaScript files, and instantiate the desired user interface elements through HTML, CSS, and JavaScript code. The ideal place to do all this is in templates.

The naive approach to integrating Bootstrap with the application is to make all the necessary changes to the HTML templates, following the recommendations given by the Bootstrap documentation. But this is an area where the use of a Flask *extension* makes an integration task much simpler, while helping keep these changes nicely organized.

The extension is called Flask-Bootstrap, and it can be installed with *pip*:

```
(venv) $ pip install flask-bootstrap
```

Flask extensions are initialized at the same time the application instance is created. Example 3-4 shows the initialization of Flask-Bootstrap.

Example 3-4. hello.py: Flask-Bootstrap initialization

```
from flask_bootstrap import Bootstrap
# ...
bootstrap = Bootstrap(app)
```

The extension is usually imported from a `flask_<name>` package, where `<name>` is the extension name. Most Flask extensions follow one of two consistent patterns for initialization. In Example 3-4, the extension is initialized by passing the application instance as an argument in the constructor. You will learn about a more advanced method to initialize extensions appropriate for larger applications in Chapter 7.

Once Flask-Bootstrap is initialized, a base template that includes all the Bootstrap files and general structure is available to the application. The application then takes advantage of Jinja2's template inheritance to extend this base template. Example 3-5 shows a new version of *user.html* as a derived template.

Example 3-5. templates/user.html: template that uses Flask-Bootstrap

```
{% extends "bootstrap/base.html" %}

{% block title %}Flasky{% endblock %}

{% block navbar %}
<div class="navbar navbar-inverse" role="navigation">
    <div class="container">
        <div class="navbar-header">
            <button type="button" class="navbar-toggle"
             data-toggle="collapse" data-target=".navbar-collapse">
                <span class="sr-only">Toggle navigation</span>
                <span class="icon-bar"></span>
                <span class="icon-bar"></span>
                <span class="icon-bar"></span>
            </button>
            <a class="navbar-brand" href="/">Flasky</a>
        </div>
        <div class="navbar-collapse collapse">
            <ul class="nav navbar-nav">
                <li><a href="/">Home</a></li>
            </ul>
        </div>
    </div>
</div>
{% endblock %}

{% block content %}
<div class="container">
    <div class="page-header">
        <h1>Hello, {{ name }}!</h1>
    </div>
```

```
</div>
{% endblock %}
```

The Jinja2 extends directive implements the template inheritance by referencing *bootstrap/base.html* from Flask-Bootstrap. The base template from Flask-Bootstrap provides a skeleton web page that includes all the Bootstrap CSS and JavaScript files.

The *user.html* template defines three blocks called title, navbar, and content. These are all blocks that the base template exports for derived templates to define. The title block is straightforward; its contents will appear between <title> tags in the header of the rendered HTML document. The navbar and content blocks are reserved for the page navigation bar and main content.

In this template, the navbar block defines a simple navigation bar using Bootstrap components. The content block has a container <div> with a page header inside. The greeting line that was in the previous version of the template is now inside the page header. Figure 3-1 shows how the application looks with these changes.

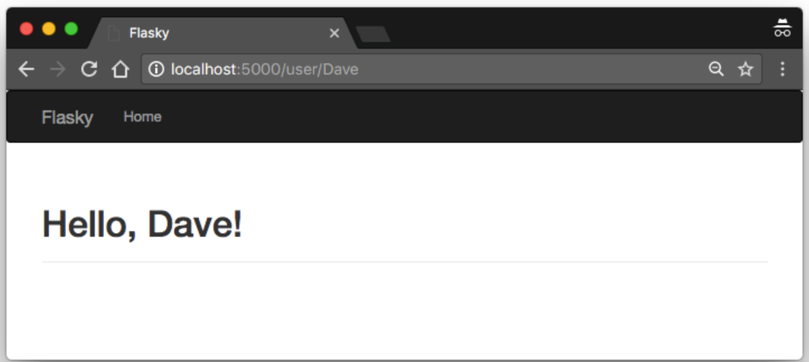


Figure 3-1. Bootstrap templates

> If you have cloned the application's Git repository on GitHub, you can run git checkout 3b to check out this version of the application. The Flask-Bootstrap package also needs to be installed in your virtual environment. The Bootstrap official documentation is a great learning resource full of copy/paste-ready examples.

Flask-Bootstrap's *base.html* template defines several other blocks that can be used in derived templates. Table 3-2 shows the complete list of available blocks.

Table 3-2. Flask-Bootstrap's base template blocks

Block name	Description
doc	The entire HTML document
html_attribs	Attributes inside the <html> tag
html	The contents of the <html> tag
head	The contents of the <head> tag
title	The contents of the <title> tag
metas	The list of <meta> tags
styles	CSS definitions
body_attribs	Attributes inside the <body> tag
body	The contents of the <body> tag
navbar	User-defined navigation bar
content	User-defined page content
scripts	JavaScript declarations at the bottom of the document

Many of the blocks in Table 3-2 are used by Flask-Bootstrap itself, so overriding them directly would cause problems. For example, the `styles` and `scripts` blocks are where the Bootstrap CSS and JavaScript files are declared. If the application needs to add its own content to a block that already has some content, then Jinja2's `super()` function must be used. For example, this is how the `scripts` block would need to be written in the derived template to add a new JavaScript file to the document:

```
{% block scripts %}
{{ super() }}
<script type="text/javascript" src="my-script.js"></script>
{% endblock %}
```

Custom Error Pages

When you enter an invalid route in your browser's address bar, you get a code 404 error page. Compared to the Bootstrap-powered pages, the default error page is now too plain and unattractive, and it has no consistency with the actual pages generated by the application.

Flask allows an application to define custom error pages that can be based on templates, like regular routes. The two most common error codes are 404, triggered when the client requests a page or route that is not known, and 500, triggered when there is an unhandled exception in the application. Example 3-6 shows how to provide custom handlers for these two errors using the `app.errorhandler` decorator.

Example 3-6. hello.py: custom error pages

```
@app.errorhandler(404)
def page_not_found(e):
    return render_template('404.html'), 404

@app.errorhandler(500)
def internal_server_error(e):
    return render_template('500.html'), 500
```

Error handlers return a response, like view functions, but they also need to return the numeric status code that corresponds to the error, which Flask conveniently accepts as a second return value.

The templates referenced in the error handlers need to be written. These templates should follow the same layout as the regular pages, so in this case they will have a navigation bar and a page header that shows the error message.

The straightforward way to write these templates is to copy *templates/user.html* to *templates/404.html* and *templates/500.html* and then change the page header elements in these two new files to the appropriate error messages, but this will generate a lot of duplication.

Jinja2's template inheritance can help with this. In the same way Flask-Bootstrap provides a base template with the basic layout of the page, the application can define its own base template with a uniform page layout that includes the navigation bar and leaves the page content to be defined in derived templates. Example 3-7 shows *templates/base.html*, a new template that inherits from *bootstrap/base.html* and defines the navigation bar but is itself a second-level base template to other templates such as *templates/user.html*, *templates/404.html*, and *templates/500.html*.

Example 3-7. templates/base.html: base application template with navigation bar

```
{% extends "bootstrap/base.html" %}

{% block title %}Flasky{% endblock %}

{% block navbar %}
<div class="navbar navbar-inverse" role="navigation">
    <div class="container">
        <div class="navbar-header">
            <button type="button" class="navbar-toggle"
             data-toggle="collapse" data-target=".navbar-collapse">
                <span class="sr-only">Toggle navigation</span>
                <span class="icon-bar"></span>
                <span class="icon-bar"></span>
                <span class="icon-bar"></span>
            </button>
            <a class="navbar-brand" href="/">Flasky</a>
```

```
        </div>
        <div class="navbar-collapse collapse">
            <ul class="nav navbar-nav">
                <li><a href="/">Home</a></li>
            </ul>
        </div>
    </div>
</div>
{% endblock %}

{% block content %}
<div class="container">
    {% block page_content %}{% endblock %}
</div>
{% endblock %}
```

The `content` block of this template is just a container `<div>` element that wraps a new empty block called `page_content`, which derived templates can define.

The templates of the application will now inherit from this template instead of directly from Flask-Bootstrap. Example 3-8 shows how simple it is to construct a custom code 404 error page that inherits from *templates/base.html*. The page for the 500 error is similar, and you can find it in the GitHub repository for the application.

Example 3-8. templates/404.html: custom code 404 error page using template inheritance

```
{% extends "base.html" %}

{% block title %}Flasky - Page Not Found{% endblock %}

{% block page_content %}
<div class="page-header">
    <h1>Not Found</h1>
</div>
{% endblock %}
```

Figure 3-2 shows how the error page looks in the browser.

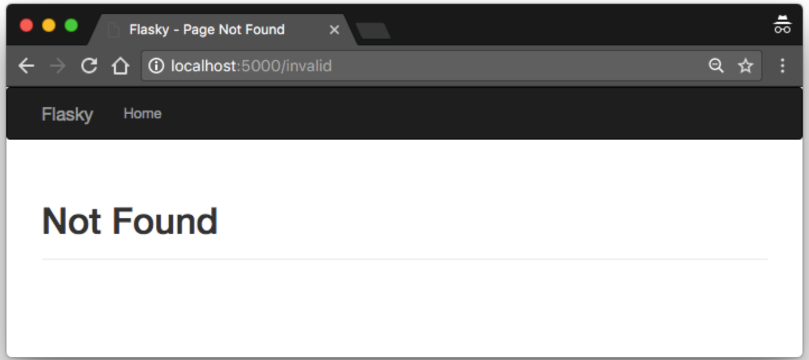


Figure 3-2. Custom code 404 error page

The *templates/user.html* template can now be simplified by making it inherit from the base template, as shown in Example 3-9.

Example 3-9. templates/user.html: simplified page template using template inheritance

```
{% extends "base.html" %}

{% block title %}Flasky{% endblock %}

{% block page_content %}
<div class="page-header">
    <h1>Hello, {{ name }}!</h1>
</div>
{% endblock %}
```

If you have cloned the application's Git repository on GitHub, you can run `git checkout 3c` to check out this version of the application.

Links

Any application that has more than one route will invariably need to include links that connect the different pages, such as in a navigation bar.

Writing the URLs as links directly in the template is trivial for simple routes, but for dynamic routes with variable portions it can get more complicated to build the URLs

right in the template. Also, URLs written explicitly create an unwanted dependency on the routes defined in the code. If the routes are reorganized, links in templates may break.

To avoid these problems, Flask provides the url_for() helper function, which generates URLs from the information stored in the application's URL map.

In its simplest usage, this function takes the view function name (or *endpoint* name for routes defined with app.add_url_route()) as its single argument and returns its URL. For example, in the current version of *hello.py* the call url_for('index') would return /, the root URL of the application. Calling url_for('index', _external=True) would instead return an absolute URL, which in this example is http://localhost:5000/.

> Relative URLs are sufficient when generating links that connect the different routes of the application. Absolute URLs are necessary only for links that will be used outside of the web browser, such as when sending links by email.

Dynamic URLs can be generated with url_for() by passing the dynamic parts as keyword arguments. For example, url_for('user', name='john', _external=True) would return http://localhost:5000/user/john.

Keyword arguments sent to url_for() are not limited to arguments used by dynamic routes. The function will add any arguments that are not dynamic to the query string. For example, url_for('user', name='john', page=2, version=1) would return /user/john?page=2&version=1.

Static Files

Web applications are not made of Python code and templates alone. Most applications also use static files such as images, JavaScript source files, and CSS files that are all referenced from the HTML code in templates.

You may recall that when the *hello.py* application's URL map was inspected in Chapter 2, a static entry appeared in it. Flask automatically supports static files by adding a special route to the application defined as /static/<filename>. For example, a call to url_for('static', filename='css/styles.css', _external=True) would return http://localhost:5000/static/css/styles.css.

In its default configuration, Flask looks for static files in a subdirectory called *static* located in the application's root folder. Files can be organized in subdirectories inside this folder if desired. When the server receives a URL that maps to a static route, it

generates a response that includes the contents of the corresponding file in the file system.

Example 3-10 shows how the application can include a *favicon.ico* icon in the base template for browsers to show in the address bar.

Example 3-10. templates/base.html: favicon definition

```
{% block head %}
{{ super() }}
<link rel="shortcut icon" href="{{ url_for('static', filename='favicon.ico') }}"
    type="image/x-icon">
<link rel="icon" href="{{ url_for('static', filename='favicon.ico') }}"
    type="image/x-icon">
{% endblock %}
```

The icon declaration is inserted at the end of the `head` block. Note how `super()` is used to preserve the original contents of the block defined in the base templates.

> If you have cloned the application's Git repository on GitHub, you can run `git checkout 3d` to check out this version of the application.

Localization of Dates and Times with Flask-Moment

Handling of dates and times in a web application is not a trivial problem when users work in different parts of the world.

The server needs uniform time units that are independent of the location of each user, so typically Coordinated Universal Time (UTC) is used. For users, however, seeing times expressed in UTC can be confusing, as users always expect to see dates and times presented in their local time and formatted according to the customs of their region.

An elegant solution that allows the server to work exclusively in UTC is to send these time units to the web browser, where they are converted to local time and rendered using JavaScript. Web browsers can do a much better job at this task because they have access to time zone and locale settings on the user's computer.

There is an excellent open source library written in JavaScript that renders dates and times in the browser called Moment.js. Flask-Moment is an extension for Flask applications that makes the integration of Moment.js into Jinja2 templates very easy. Flask-Moment is installed with *pip*:

```
(venv) $ pip install flask-moment
```

The extension is initialized in a similar way to Flask-Bootstrap. The required code is shown in Example 3-11.

Example 3-11. hello.py: initializing Flask-Moment

```
from flask_moment import Moment
moment = Moment(app)
```

Flask-Moment depends on jQuery.js in addition to Moment.js. These two libraries need to be included somewhere in the HTML document—either directly, in which case you can choose what versions to use, or through the helper functions provided by the extension, which reference tested versions of these libraries from a content delivery network (CDN). Because Bootstrap already includes jQuery.js, only Moment.js needs to be added in this case. Example 3-12 shows how this library is loaded in the `scripts` block of the template, while also preserving the original contents of the block provided by the base template. Note that since this is a predefined block in the Flask-Bootstrap base template, the location in *templates/base.html* where this block is inserted does not matter.

Example 3-12. templates/base.html: importing the Moment.js library

```
{% block scripts %}
{{ super() }}
{{ moment.include_moment() }}
{% endblock %}
```

To work with timestamps, Flask-Moment makes a `moment` object available to templates. Example 3-13 demonstrates passing a variable called `current_time` to the template for rendering.

Example 3-13. hello.py: adding a datetime variable

```
from datetime import datetime

@app.route('/')
def index():
    return render_template('index.html',
                           current_time=datetime.utcnow())
```

Example 3-14 shows how this `current_time` template variable is rendered.

Example 3-14. templates/index.html: timestamp rendering with Flask-Moment

```
<p>The local date and time is {{ moment(current_time).format('LLL') }}.</p>
<p>That was {{ moment(current_time).fromNow(refresh=True) }}</p>
```

The `format('LLL')` function renders the date and time according to the time zone and locale settings in the client computer. The argument determines the rendering style, from `'L'` to `'LLLL'` for four different levels of verbosity. The `format()` function can also accept a long list of custom format specifiers.

The `fromNow()` render style shown in the second line renders a relative timestamp and automatically refreshes it as time passes. Initially this timestamp will be shown as "a few seconds ago," but the `refresh=True` option will keep it updated as time passes, so if you leave the page open for a few minutes you will see the text changing to "a minute ago," then "2 minutes ago," and so on.

Figure 3-3 shows how the *http://localhost:5000/* route looks after the two timestamps are added to the *index.html* template.

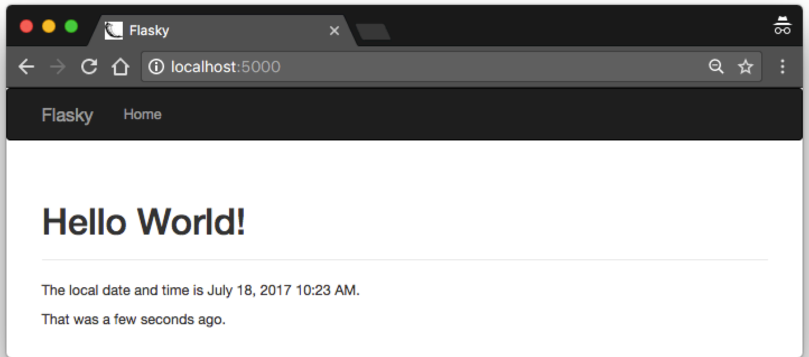


Figure 3-3. Page with two Flask-Moment timestamps

> If you have cloned the application's Git repository on GitHub, you can run `git checkout 3e` to check out this version of the application.

Flask-Moment implements the `format()`, `fromNow()`, `fromTime()`, `calendar()`, `valueOf()`, and `unix()` methods from Moment.js. Consult the Moment.js documentation to learn about all the formatting options offered by this library.

> Flask-Moment assumes that timestamps handled by the server-side application are “naive” `datetime` objects expressed in UTC. See the documentation for the `datetime` package in the standard library for information on naive and aware date and time objects.

The timestamps rendered by Flask-Moment can be localized to many languages. A language can be selected in the template by passing the two-letter language code to function `locale()`, right after the Moment.js library is included. For example, here is how to configure Moment.js to use Spanish:

```
{% block scripts %}
{{ super() }}
{{ moment.include_moment() }}
{{ moment.locale('es') }}
{% endblock %}
```

With all the techniques discussed in this chapter, you should be able to build modern and user-friendly web pages for your application. The next chapter touches on an aspect of templates not yet discussed: how to interact with the user through web forms.

CHAPTER 4

Web Forms

The templates that you worked with in Chapter 3 are unidirectional, in the sense that they allow information to flow from the server to the user. For most applications, however, there is also a need to have information that flows in the other direction, with the user providing data that the server accepts and processes.

With HTML, it is possible to create web forms, in which users can enter information. The form data is then submitted by the web browser to the server, typically in the form of a `POST` request. The Flask request object, introduced in Chapter 2, exposes all the information sent by the client in a request and, in particular for `POST` requests containing form data, provides access to the user information through `request.form`.

Although the support provided in Flask's request object is sufficient for the handling of web forms, there are a number of tasks that can become tedious and repetitive. Two good examples are the generation of HTML code for the forms and the validation of the submitted form data.

The Flask-WTF extension makes working with web forms a much more pleasant experience. This extension is a Flask integration wrapper around the framework-agnostic WTForms package.

Flask-WTF and its dependencies can be installed with *pip*:

```
(venv) $ pip install flask-wtf
```

Configuration

Unlike most other extensions, Flask-WTF does not need to be initialized at the application level, but it expects the application to have a *secret key* configured. A secret key is a string with any random and unique content that is used as an encryption or signing key to improve the security of the application in several ways. Flask uses this key to protect the contents of the user session against tampering. You should pick a different secret key in each application that you build and make sure that this string is not known by anyone. Example 4-1 shows how to configure a secret key in a Flask application.

Example 4-1. hello.py: Flask-WTF configuration

```
app = Flask(__name__)
app.config['SECRET_KEY'] = 'hard to guess string'
```

The `app.config` dictionary is a general-purpose place to store configuration variables used by Flask, extensions, or the application itself. Configuration values can be added to the `app.config` object using standard dictionary syntax. The configuration object also has methods to import configuration values from files or the environment. A more practical way to manage configuration values for a larger application will be discussed in Chapter 7.

Flask-WTF requires a secret key to be configured in the application because this key is part of the mechanism the extension uses to protect all forms against cross-site request forgery (CSRF) attacks. A CSRF attack occurs when a malicious website sends requests to the application server on which the user is currently logged in. Flask-WTF generates security tokens for all forms and stores them in the user session, which is protected with a cryptographic signature generated from the secret key.

> For added security, the secret key should be stored in an environment variable instead of being embedded in the source code. This technique is described in Chapter 7.

Form Classes

When using Flask-WTF, each web form is represented in the server by a class that inherits from the class `FlaskForm`. The class defines the list of fields in the form, each represented by an object. Each field object can have one or more *validators* attached. A validator is a function that checks whether the data submitted by the user is valid.

Example 4-2 shows a simple web form that has a text field and a submit button.

Example 4-2. hello.py: form class definition

```
from flask_wtf import FlaskForm
from wtforms import StringField, SubmitField
from wtforms.validators import DataRequired

class NameForm(FlaskForm):
    name = StringField('What is your name?', validators=[DataRequired()])
    submit = SubmitField('Submit')
```

The fields in the form are defined as class variables, and each class variable is assigned an object associated with the field type. In this example, the `NameForm` form has a text field called `name` and a submit button called `submit`. The `StringField` class represents an HTML `<input>` element with a `type="text"` attribute. The `SubmitField` class represents an HTML `<input>` element with a `type="submit"` attribute. The first argument to the field constructors is the label that will be used when rendering the form to HTML.

The optional `validators` argument included in the `StringField` constructor defines a list of checkers that will be applied to the data submitted by the user before it is accepted. The `DataRequired()` validator ensures that the field is not submitted empty.

> The `FlaskForm` base class is defined by the Flask-WTF extension, so it is imported from `flask_wtf`. The fields and validators, however, are imported directly from the WTForms package.

The list of standard HTML fields supported by WTForms is shown in Table 4-1.

Table 4-1. WTForms standard HTML fields

Field type	Description
`BooleanField`	Checkbox with `True` and `False` values
`DateField`	Text field that accepts a `datetime.date` value in a given format
`DateTimeField`	Text field that accepts a `datetime.datetime` value in a given format
`DecimalField`	Text field that accepts a `decimal.Decimal` value
`FileField`	File upload field
`HiddenField`	Hidden text field
`MultipleFileField`	Multiple file upload field
`FieldList`	List of fields of a given type

Field type	Description
`FloatField`	Text field that accepts a floating-point value
`FormField`	Form embedded as a field in a container form
`IntegerField`	Text field that accepts an integer value
`PasswordField`	Password text field
`RadioField`	List of radio buttons
`SelectField`	Drop-down list of choices
`SelectMultipleField`	Drop-down list of choices with multiple selection
`SubmitField`	Form submission button
`StringField`	Text field
`TextAreaField`	Multiple-line text field

The list of WTForms built-in validators is shown in Table 4-2.

Table 4-2. WTForms validators

Validator	Description
`DataRequired`	Validates that the field contains data after type conversion
`Email`	Validates an email address
`EqualTo`	Compares the values of two fields; useful when requesting a password to be entered twice for confirmation
`InputRequired`	Validates that the field contains data before type conversion
`IPAddress`	Validates an IPv4 network address
`Length`	Validates the length of the string entered
`MacAddress`	Validates a MAC address
`NumberRange`	Validates that the value entered is within a numeric range
`Optional`	Allows empty input in the field, skipping additional validators
`Regexp`	Validates the input against a regular expression
`URL`	Validates a URL
`UUID`	Validates a UUID
`AnyOf`	Validates that the input is one of a list of possible values
`NoneOf`	Validates that the input is none of a list of possible values

HTML Rendering of Forms

Form fields are callables that, when invoked from a template, render themselves to HTML. Assuming that the view function passes a `NameForm` instance to the template as an argument named `form`, the template can generate a simple HTML form as follows:

```
<form method="POST">
    {{ form.hidden_tag() }}
    {{ form.name.label }} {{ form.name() }}
    {{ form.submit() }}
</form>
```

Note that in addition to the `name` and `submit` fields, the form has a `form.hidden_tag()` element. This element defines an extra form field that is hidden, used by Flask-WTF to implement CSRF protection.

Of course, the result of rendering a web form in this way is extremely bare. Any keyword arguments added to the calls that render the fields are converted into HTML attributes for the field—so, for example, you can give the field `id` or `class` attributes and then define CSS styles for them:

```
<form method="POST">
    {{ form.hidden_tag() }}
    {{ form.name.label }} {{ form.name(id='my-text-field') }}
    {{ form.submit() }}
</form>
```

But even with HTML attributes, the effort required to render a form in this way and make it look good is significant, so it is best to leverage Bootstrap's own set of form styles whenever possible. The Flask-Bootstrap extension provides a high-level helper function that renders an entire Flask-WTF form using Bootstrap's predefined form styles, all with a single call. Using Flask-Bootstrap, the previous form can be rendered as follows:

```
{% import "bootstrap/wtf.html" as wtf %}
{{ wtf.quick_form(form) }}
```

The `import` directive works in the same way as regular Python scripts do and allows template elements to be imported and used in many templates. The imported *bootstrap/wtf.html* file defines helper functions that render Flask-WTF forms using Bootstrap. The `wtf.quick_form()` function takes a Flask-WTF form object and renders it using default Bootstrap styles. The complete template for *hello.py* is shown in Example 4-3.

Example 4-3. templates/index.html: using Flask-WTF and Flask-Bootstrap to render a form

```
{% extends "base.html" %}
{% import "bootstrap/wtf.html" as wtf %}

{% block title %}Flasky{% endblock %}

{% block page_content %}
<div class="page-header">
    <h1>Hello, {% if name %}{{ name }}{% else %}Stranger{% endif %}!</h1>
</div>
{{ wtf.quick_form(form) }}
{% endblock %}
```

The content area of the template now has two sections. The first section is a page header that shows a greeting. Here a template conditional is used. Conditionals in Jinja2 have the format `{% if condition %}...{% else %}...{% endif %}`. If the condition evaluates to `True`, then what appears between the `if` and `else` directives is added to the rendered template. If the condition evaluates to `False`, then what's between the `else` and `endif` is rendered instead. The purpose of this is to render `Hello, {{ name }}!` when the `name` template variable is defined, or the string `Hello, Stranger!` when it is not. The second section of the content renders the `NameForm` form using the `wtf.quick_form()` function.

Form Handling in View Functions

In the new version of *hello.py*, the `index()` view function will have two tasks. First it will render the form, and then it will receive the form data entered by the user. Example 4-4 shows the updated `index()` view function.

Example 4-4. hello.py: handle a web form with GET and POST request methods

```
@app.route('/', methods=['GET', 'POST'])
def index():
    name = None
    form = NameForm()
    if form.validate_on_submit():
        name = form.name.data
        form.name.data = ''
    return render_template('index.html', form=form, name=name)
```

The `methods` argument added to the `app.route` decorator tells Flask to register the view function as a handler for `GET` and `POST` requests in the URL map. When `methods` is not given, the view function is registered to handle `GET` requests only.

Adding `POST` to the method list is necessary because form submissions are much more conveniently handled as `POST` requests. It is possible to submit a form as a `GET` request, but as `GET` requests have no body, the data is appended to the URL as a query string and becomes visible in the browser's address bar. For this and several other reasons, form submissions are almost universally done as `POST` requests.

The local `name` variable is used to hold the name received from the form when available; when the name is not known, the variable is initialized to `None`. The view function creates an instance of the `NameForm` class shown previously to represent the form. The `validate_on_submit()` method of the form returns `True` when the form was submitted and the data was accepted by all the field validators. In all other cases, `validate_on_submit()` returns `False`. The return value of this method effectively serves to determine whether the form needs to be rendered or processed.

When a user navigates to the application for the first time, the server will receive a `GET` request with no form data, so `validate_on_submit()` will return `False`. The body of the `if` statement will be skipped and the request will be handled by rendering the template, which gets the form object and the `name` variable set to `None` as arguments. Users will now see the form displayed in the browser.

When the form is submitted by the user, the server receives a `POST` request with the data. The call to `validate_on_submit()` invokes the `DataRequired()` validator attached to the name field. If the name is not empty, then the validator accepts it and `validate_on_submit()` returns `True`. Now the name entered by the user is accessible as the `data` attribute of the field. Inside the body of the `if` statement, this name is assigned to the local `name` variable and the form field is cleared by setting that `data` attribute to an empty string, so that the field is blanked when the form is rendered to the page again. The `render_template()` call in the last line renders the template, but this time the `name` argument contains the name from the form, so the greeting will be personalized.

> If you have cloned the application's Git repository on GitHub, you can run `git checkout 4a` to check out this version of the application.

Figure 4-1 shows how the form looks in the browser window when a user initially enters the site. When the user submits a name, the application responds with a personalized greeting. The form still appears below it, so a user can submit it multiple times with different names if desired. Figure 4-2 shows the application in this state.

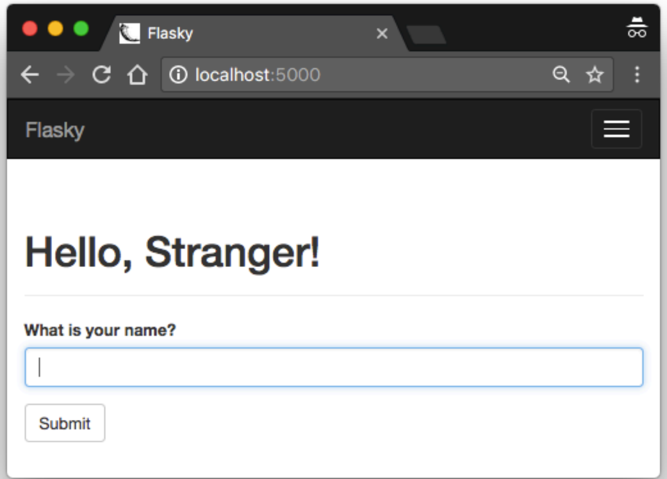


Figure 4-1. Flask-WTF web form

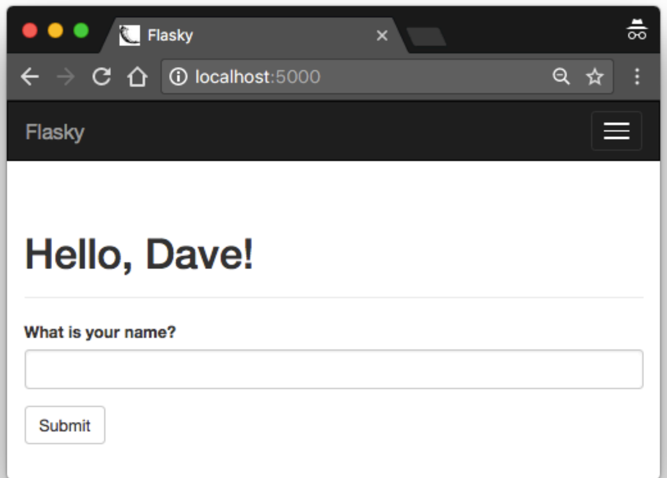


Figure 4-2. Web form after submission

If the user submits the form with an empty name, the `DataRequired()` validator catches the error, as seen in Figure 4-3. Note how much functionality is being provided automatically. This is a great example of the power that well-designed extensions like Flask-WTF and Flask-Bootstrap can give to your application.

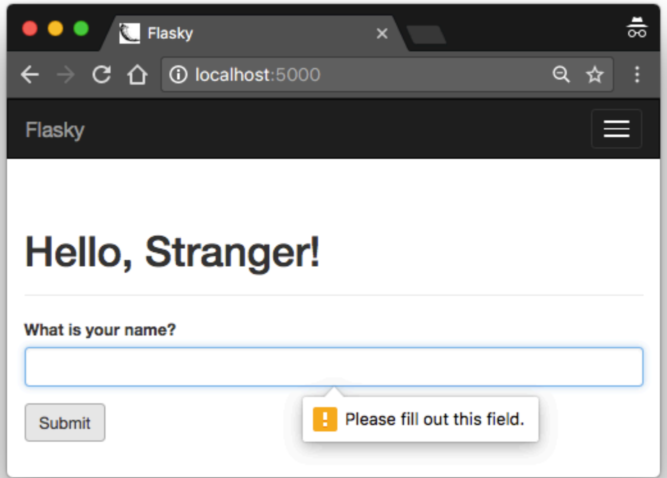


Figure 4-3. Web form after failed validator

Redirects and User Sessions

The last version of *hello.py* has a usability problem. If you enter your name and submit it, and then click the refresh button in your browser, you will likely get an obscure warning that asks for confirmation before submitting the form again. This happens because browsers repeat the last request they sent when they are asked to refresh a page. When the last request sent is a `POST` request with form data, a refresh would cause a duplicate form submission, which in almost all cases is not the desired action. For that reason, the browser asks for confirmation from the user.

Many users do not understand this warning from the browser. Consequently, it is considered good practice for web applications to never leave a `POST` request as the last request sent by the browser.

This is achieved by responding to `POST` requests with a *redirect* instead of a normal response. A redirect is a special type of response that contains a URL instead of a string with HTML code. When the browser receives a redirect response, it issues a

GET request for the redirect URL, and that is the page that it displays. The page may take a few more milliseconds to load because of the second request that has to be sent to the server, but other than that, the user will not see any difference. Now the last request is a GET, so the refresh command works as expected. This trick is known as the *Post/Redirect/Get pattern*.

But this approach brings a second problem. When the application handles the POST request, it has access to the name entered by the user in `form.name.data`, but as soon as that request ends the form data is lost. Because the POST request is handled with a redirect, the application needs to store the name so that the redirected request can have it and use it to build the actual response.

Applications can "remember" things from one request to the next by storing them in the *user session*, a private storage that is available to each connected client. The user session was introduced in Chapter 2 as one of the variables associated with the request context. It's called `session` and is accessed like a standard Python dictionary.

> By default, user sessions are stored in client-side cookies that are cryptographically signed using the configured secret key. Any tampering with the cookie content would render the signature invalid, thus invalidating the session.

Example 4-5 shows a new version of the `index()` view function that implements redirects and user sessions.

Example 4-5. hello.py: redirects and user sessions

```
from flask import Flask, render_template, session, redirect, url_for

@app.route('/', methods=['GET', 'POST'])
def index():
    form = NameForm()
    if form.validate_on_submit():
        session['name'] = form.name.data
        return redirect(url_for('index'))
    return render_template('index.html', form=form, name=session.get('name'))
```

In the previous version of the application, a local `name` variable was used to store the name entered by the user in the form. That variable is now placed in the user session as `session['name']` so that it is remembered beyond the request.

Requests that come with valid form data will now end with a call to `redirect()`, a Flask helper function that generates the HTTP redirect response. The `redirect()` function takes the URL to redirect to as an argument. The redirect URL used in this case is the root URL, so the response could have been written more concisely as `redirect('/')`, but instead Flask's URL generator function `url_for()`, introduced in Chapter 3, is used.

The first and only required argument to `url_for()` is the *endpoint* name, the internal name each route has. By default, the endpoint of a route is the name of the view function attached to it. In this example, the view function that handles the root URL is `index()`, so the name given to `url_for()` is `index`.

The last change is in the `render_template()` function, which now obtains the `name` argument directly from the session using `session.get('name')`. As with regular dictionaries, using `get()` to request a dictionary key avoids an exception for keys that aren't found. The `get()` method returns a default value of `None` for a missing key.

> If you have cloned the application's Git repository on GitHub, you can run `git checkout 4b` to check out this version of the application.

With this version of the application, you can see that refreshing the page in your browser always results in the expected behavior.

Message Flashing

Sometimes it is useful to give the user a status update after a request is completed. This could be a confirmation message, a warning, or an error. A typical example is when you submit a login form to a website with a mistake and the server responds by rendering the login form again with a message above it that informs you that your username or password is invalid.

Flask includes this functionality as a core feature. Example 4-6 shows how the `flash()` function can be used for this purpose.

Example 4-6. hello.py: flashed messages

```
from flask import Flask, render_template, session, redirect, url_for, flash

@app.route('/', methods=['GET', 'POST'])
def index():
    form = NameForm()
    if form.validate_on_submit():
        old_name = session.get('name')
        if old_name is not None and old_name != form.name.data:
            flash('Looks like you have changed your name!')
        session['name'] = form.name.data
        return redirect(url_for('index'))
    return render_template('index.html',
        form = form, name = session.get('name'))
```

In this example, each time a name is submitted it is compared against the name stored in the user session, which will have been put there during a previous submission of the same form. If the two names are different, the `flash()` function is invoked with a message to be displayed on the next response sent back to the client.

Calling `flash()` is not enough to get messages displayed; the templates used by the application need to render these messages. The best place to render flashed messages is the base template, because that will enable these messages in all pages. Flask makes a `get_flashed_messages()` function available to templates to retrieve the messages and render them, as shown in Example 4-7.

Example 4-7. templates/base.html: rendering of flashed messages

```
{% block content %}
<div class="container">
    {% for message in get_flashed_messages() %}
    <div class="alert alert-warning">
        <button type="button" class="close" data-dismiss="alert">&times;</button>
        {{ message }}
    </div>
    {% endfor %}

    {% block page_content %}{% endblock %}
</div>
{% endblock %}
```

In this example, messages are rendered using Bootstrap's alert CSS styles for warning messages (one is shown in Figure 4-4).

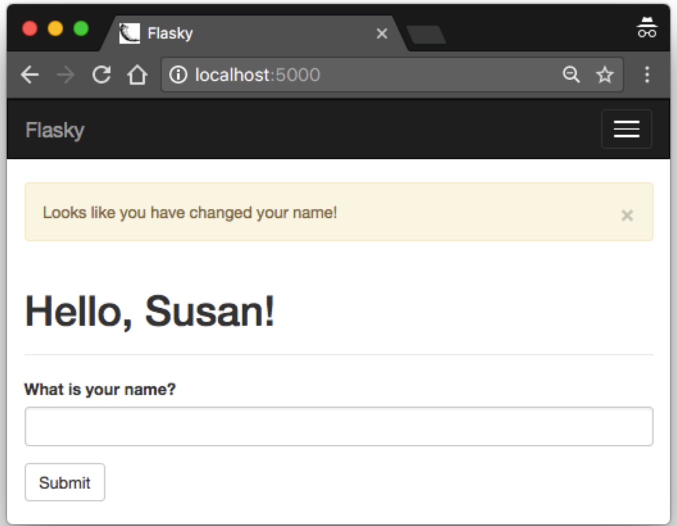


Figure 4-4. Flashed message

A loop is used because there could be multiple messages queued for display, one for each time `flash()` was called in the previous request cycle. Messages that are retrieved from `get_flashed_messages()` will not be returned the next time this function is called, so flashed messages appear only once and are then discarded.

> If you have cloned the application's Git repository on GitHub, you can run `git checkout 4c` to check out this version of the application.

Being able to accept data from the user through web forms is a feature required by most applications, and so is the ability to store that data in permanent storage. Using databases with Flask is the topic of the next chapter.

CHAPTER 5

Databases

A *database* stores application data in an organized way. The application then issues *queries* to retrieve specific portions of the data as they are needed. The most commonly used databases for web applications are those based on the *relational* model, also called SQL databases in reference to the Structured Query Language they use. But in recent years *document-oriented* and *key-value* databases, informally known together as NoSQL databases, have become popular alternatives.

SQL Databases

Relational databases store data in *tables*, which model the different entities in the application's domain. For example, a database for an order management application will likely have `customers`, `products`, and `orders` tables.

A table has a fixed number of *columns* and a variable number of *rows*. The columns define the data attributes of the entity represented by the table. For example, a `customers` table will have columns such as `name`, `address`, `phone`, and so on. Each row in a table defines an actual data element that assigns values to some or all the columns.

Tables have a special column called the *primary key*, which holds a unique identifier for each row stored in the table. Tables can also have columns called *foreign keys*, which reference the primary key of a row in the same or another table. These links between rows are called *relationships* and are the foundation of the relational database model.

Figure 5-1 shows a diagram of a simple database with two tables that store users and user roles. The line that connects the two tables represents a relationship between the tables.

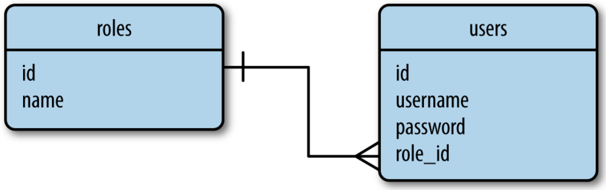


Figure 5-1. Relational database example

This graphical style of representing the structure of a database is called an *entity-relationship diagram*. In this representation, boxes represent database tables, showing lists of the table's attributes or columns. The `roles` table stores the list of all possible user roles, each identified by a unique `id` value—the table's primary key. The `users` table contains the list of users, each with its own unique `id` as well. Besides the `id` primary keys, the `roles` table has a `name` column and the `users` table has `username` and `password` columns.

The `role_id` column in the `users` table is a *foreign key*. The line that connects the `roles.id` and `users.role_id` columns represents a relationship between the two tables. The symbols attached to the line at each end indicate the *cardinality* of the relationship. On the `roles.id` side, the line is shown to have a "one," while on the `users.role_id` side a "many" is represented. This depicts a *one-to-many* relationship, indicating that each row from the `roles` table can be associated with many rows from the `users` table.

As seen in the example, relational databases store data efficiently and avoid duplication. Renaming a user role in this database is simple because role names exist in a single place. Immediately after a role name is changed in the `roles` table, all users that have a `role_id` that references the changed role will see the update.

On the other hand, having the data split into multiple tables can be a complication. Producing a listing of users with their roles presents a small problem, because users and user roles need to be read from two tables and *joined* before they can be presented together. Relational database engines provide the support to perform join operations between tables when necessary.

NoSQL Databases

Databases that do not follow the relational model described in the previous section are collectively referred to as *NoSQL* databases. One common organization for NoSQL databases uses *collections* instead of tables and *documents* instead of records. NoSQL databases are designed in a way that makes joins difficult, so most of them do not support this operation at all. For a NoSQL database structured as in Figure 5-1,

listing the users with their roles requires the application itself to perform the join operation by reading the `role_id` field of each user and then searching the `roles` table for it.

A more appropriate design for a NoSQL database is shown in Figure 5-2. This is the result of applying an operation called *denormalization*, which reduces the number of tables at the expense of data duplication.

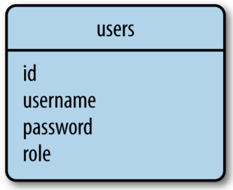


Figure 5-2. NoSQL database example

A database with this structure has the role name explicitly stored with each user. Renaming a role can then turn out to be an expensive operation that may require updating a large number of documents.

But it isn't all bad news with NoSQL databases. Having the data duplicated allows for faster querying. Listing users and their roles is straightforward because no joins are needed.

SQL or NoSQL?

SQL databases excel at storing structured data in an efficient and compact form. These databases go to great lengths to preserve consistency, even in the face of power failures or hardware malfunctions. The paradigm that allows relational databases to reach this high level of reliability is called ACID, which stands for Atomicity, Consistency, Isolation, and Durability. NoSQL databases relax some of the ACID requirements and as a result can sometimes get a performance edge.

A full analysis and comparison of database types is outside the scope of this book. For small to medium-sized applications, both SQL and NoSQL databases are perfectly capable and have practically equivalent performance.

Python Database Frameworks

Python has packages for most database engines, both open source and commercial. Flask puts no restrictions on what database packages can be used, so you can work with MySQL, Postgres, SQLite, Redis, MongoDB, CouchDB, or DynamoDB if any of these is your favorite.

As if those weren't enough choices, there are also a number of database abstraction layer packages, such as SQLAlchemy or MongoEngine, that allow you to work at a higher level with regular Python objects instead of database entities such as tables, documents, or query languages.

There are a number of factors to evaluate when choosing a database framework:

Ease of use
: When comparing straight database engines to database abstraction layers, the second group clearly wins. Abstraction layers, also called object-relational mappers (ORMs) or object-document mappers (ODMs), provide transparent conversion of high-level object-oriented operations into low-level database instructions.

Performance
: The conversions that ORMs and ODMs have to do to translate from the object domain into the database domain have an overhead. In most cases, the performance penalty is negligible, but it may not always be. In general, the productivity gain obtained with ORMs and ODMs far outweighs a minimal performance degradation, so this isn't a valid argument to drop ORMs and ODMs completely. What makes sense is to choose a database abstraction layer that provides optional access to the underlying database in case specific operations need to be optimized by implementing them directly as native database instructions.

Portability
: The database choices available on your development and production platforms must be considered. For example, if you plan to host your application on a cloud platform, then you should find out what database choices this service offers.

  Another portability aspect applies to ORMs and ODMs. Although some of these frameworks provide an abstraction layer for a single database engine, others abstract even higher and provide a choice of database engines—all accessible with the same object-oriented interface. The best example of this is the SQLAlchemy ORM, which supports a list of relational database engines including the popular MySQL, Postgres, and SQLite.

Flask integration
: Choosing a framework that has integration with Flask is not absolutely required, but it will save you from having to write the integration code yourself. Flask integration could simplify configuration and operation, so using a package specifically designed as a Flask extension should be preferred.

Based on these goals, the chosen database framework for the examples in this book will be Flask-SQLAlchemy, the Flask extension wrapper for SQLAlchemy.

Database Management with Flask-SQLAlchemy

Flask-SQLAlchemy is a Flask extension that simplifies the use of SQLAlchemy inside Flask applications. SQLAlchemy is a powerful relational database framework that supports several database backends. It offers a high-level ORM and low-level access to the database's native SQL functionality.

Like most other extensions, Flask-SQLAlchemy is installed with *pip*:

```
(venv) $ pip install flask-sqlalchemy
```

In Flask-SQLAlchemy, a database is specified as a URL. Table 5-1 lists the format of the URLs for the three most popular database engines.

Table 5-1. Flask-SQLAlchemy database URLs

Database engine	URL
MySQL	*mysql://username:password@hostname/database*
Postgres	*postgresql://username:password@hostname/database*
SQLite (Linux, macOS)	*sqlite:////absolute/path/to/database*
SQLite (Windows)	*sqlite:///c:/absolute/path/to/database*

In these URLs, *hostname* refers to the server that hosts the database service, which could be *localhost* or a remote server. Database servers can host several databases, so *database* indicates the name of the database to use. For databases that need authentication, *username* and *password* are the database user credentials.

SQLite databases do not have a server, so *hostname*, *username*, and *password* are omitted and *database* is the filename on disk for the database.

The URL of the application database must be configured as the key `SQLALCHEMY_DATABASE_URI` in the Flask configuration object. The Flask-SQLAlchemy documentation also suggests setting key `SQLALCHEMY_TRACK_MODIFICATIONS` to `False` to use less memory unless signals for object changes are needed. Consult the Flask-SQLAlchemy documentation for information on other configuration options. Example 5-1 shows how to initialize and configure a simple SQLite database.

Example 5-1. hello.py: database configuration

```python
import os
from flask_sqlalchemy import SQLAlchemy

basedir = os.path.abspath(os.path.dirname(__file__))
```

```
app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] =\
    'sqlite:///' + os.path.join(basedir, 'data.sqlite')
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False

db = SQLAlchemy(app)
```

The `db` object instantiated from the class `SQLAlchemy` represents the database and provides access to all the functionality of Flask-SQLAlchemy.

Model Definition

The term *model* is used when referring to the persistent entities used by the application. In the context of an ORM, a model is typically a Python class with attributes that match the columns of a corresponding database table.

The database instance from Flask-SQLAlchemy provides a base class for models as well as a set of helper classes and functions that are used to define their structure. The `roles` and `users` tables from Figure 5-1 can be defined as the models `Role` and `User` as shown in Example 5-2.

Example 5-2. hello.py: Role and User model definition

```
class Role(db.Model):
    __tablename__ = 'roles'
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(64), unique=True)

    def __repr__(self):
        return '<Role %r>' % self.name

class User(db.Model):
    __tablename__ = 'users'
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(64), unique=True, index=True)

    def __repr__(self):
        return '<User %r>' % self.username
```

The `__tablename__` class variable defines the name of the table in the database. Flask-SQLAlchemy assigns a default table name if `__tablename__` is omitted, but those default names do not follow the popular convention of using plurals for table names, so it is best to name tables explicitly. The remaining class variables are the attributes of the model, defined as instances of the `db.Column` class.

The first argument given to the `db.Column` constructor is the type of the database column and model attribute. Table 5-2 lists some of the column types that are available, along with the Python types used in the model.

Table 5-2. Most common SQLAlchemy column types

Type name	Python type	Description
`Integer`	`int`	Regular integer, typically 32 bits
`SmallInteger`	`int`	Short-range integer, typically 16 bits
`BigInteger`	`int` or `long`	Unlimited precision integer
`Float`	`float`	Floating-point number
`Numeric`	`decimal.Decimal`	Fixed-point number
`String`	`str`	Variable-length string
`Text`	`str`	Variable-length string, optimized for large or unbounded length
`Unicode`	`unicode`	Variable-length Unicode string
`UnicodeText`	`unicode`	Variable-length Unicode string, optimized for large or unbounded length
`Boolean`	`bool`	Boolean value
`Date`	`datetime.date`	Date value
`Time`	`datetime.time`	Time value
`DateTime`	`datetime.datetime`	Date and time value
`Interval`	`datetime.timedelta`	Time interval
`Enum`	`str`	List of string values
`PickleType`	Any Python object	Automatic Pickle serialization
`LargeBinary`	`str`	Binary blob

The remaining arguments to `db.Column` specify configuration options for each attribute. Table 5-3 lists some of the options available.

Table 5-3. Most common SQLAlchemy column options

Option name	Description
`primary_key`	If set to `True`, the column is the table's primary key.
`unique`	If set to `True`, do not allow duplicate values for this column.
`index`	If set to `True`, create an index for this column, so that queries are more efficient.
`nullable`	If set to `True`, allow empty values for this column. If set to `False`, the column will not allow null values.
`default`	Define a default value for the column.

Flask-SQLAlchemy requires all models to define a primary key column, which is commonly named `id`.

Although it's not strictly necessary, the two models include a `__repr__()` method to give them a readable string representation that can be used for debugging and testing purposes.

Relationships

Relational databases establish connections between rows in different tables through the use of relationships. The relational diagram in Figure 5-1 expresses a simple relationship between users and their roles. This is a *one-to-many* relationship from roles to users, because one role can belong to many users, but each user can have only one role.

Example 5-3 shows how the one-to-many relationship in Figure 5-1 is represented in the model classes.

Example 5-3. hello.py: relationships in the database models

```
class Role(db.Model):
    # ...
    users = db.relationship('User', backref='role')

class User(db.Model):
    # ...
    role_id = db.Column(db.Integer, db.ForeignKey('roles.id'))
```

As seen in Figure 5-1, a relationship connects two rows through the use of a foreign key. The `role_id` column added to the `User` model is defined as a foreign key, and that establishes the relationship. The `'roles.id'` argument to `db.ForeignKey()` specifies that the column should be interpreted as having `id` values from rows in the `roles` table.

The `users` attribute added to the model `Role` represents the object-oriented view of the relationship, as seen from the "one" side. Given an instance of class `Role`, the `users` attribute will return the list of users associated with that role (i.e., the "many" side). The first argument to `db.relationship()` indicates what model is on the other side of the relationship. The model class can be provided as a string if the class is defined later in the module.

The `backref` argument to `db.relationship()` defines the reverse direction of the relationship, by adding a `role` attribute to the `User` model. This attribute can be used

on any instance of `User` instead of the `role_id` foreign key to access the `Role` model as an object.

In most cases `db.relationship()` can locate the relationship's foreign key on its own, but sometimes it cannot determine what column to use as a foreign key. For example, if the `User` model had two or more columns defined as `Role` foreign keys, then SQLAlchemy would not know which one of the two to use. Whenever the foreign key configuration is ambiguous, additional arguments to `db.relationship()` need to be given. Table 5-4 lists some of the common configuration options that can be used to define a relationship.

Table 5-4. Common SQLAlchemy relationship options

Option name	Description
`backref`	Add a back reference in the other model in the relationship.
`primaryjoin`	Specify the join condition between the two models explicitly. This is necessary only for ambiguous relationships.
`lazy`	Specify how the related items are to be loaded. Possible values are `select` (items are loaded on demand the first time they are accessed), `immediate` (items are loaded when the source object is loaded), `joined` (items are loaded immediately, but as a join), `subquery` (items are loaded immediately, but as a subquery), `noload` (items are never loaded), and `dynamic` (instead of loading the items, the query that can load them is given).
`uselist`	If set to `False`, use a scalar instead of a list.
`order_by`	Specify the ordering used for the items in the relationship.
`secondary`	Specify the name of the association table to use in many-to-many relationships.
`secondaryjoin`	Specify the secondary join condition for many-to-many relationships when SQLAlchemy cannot determine it on its own.

If you have cloned the application's Git repository on GitHub, you can run `git checkout 5a` to check out this version of the application.

There are other relationship types besides one-to-many. The *one-to-one* relationship can be expressed the same way as one-to-many, as described earlier, but with the `uselist` option set to `False` within the `db.relationship()` definition so that the "many" side becomes a "one" side. The *many-to-one* relationship can also be expressed as a one-to-many if the tables are reversed, or it can be expressed with the foreign key and the `db.relationship()` definition both on the "many" side. The most complex relationship type, *many-to-many*, requires an additional table called an *association* or *junction table*. You will learn about many-to-many relationships in Chapter 12.

Database Operations

The models are now fully configured according to the database diagram in Figure 5-1 and are ready to be used. The best way to learn how to work with these models is in a Python shell. The following sections will walk you through the most common database operations in a shell started with the `flask shell` command. Before you use this command, make sure the `FLASK_APP` environment variable is set to `hello.py`, as shown in Chapter 2.

Creating the Tables

The very first thing to do is to instruct Flask-SQLAlchemy to create a database based on the model classes. The `db.create_all()` function locates all the subclasses of `db.Model` and creates corresponding tables in the database for them:

```
(venv) $ flask shell
>>> from hello import db
>>> db.create_all()
```

If you check the application directory, you will now see a new file there called *data.sqlite*, the name that was given to the SQLite database in the configuration. The `db.create_all()` function will not re-create or update a database table if it already exists in the database. This can be inconvenient when the models are modified and the changes need to be applied to an existing database. The brute-force solution to update existing database tables to a different schema is to remove the old tables first:

```
>>> db.drop_all()
>>> db.create_all()
```

Unfortunately, this method has the undesired side effect of destroying all the data in the old database. A better solution to the problem of updating databases is presented near the end of the chapter.

Inserting Rows

The following example creates a few roles and users:

```
>>> from hello import Role, User
>>> admin_role = Role(name='Admin')
>>> mod_role = Role(name='Moderator')
>>> user_role = Role(name='User')
>>> user_john = User(username='john', role=admin_role)
>>> user_susan = User(username='susan', role=user_role)
>>> user_david = User(username='david', role=user_role)
```

The constructors for models accept initial values for the model attributes as keyword arguments. Note that the `role` attribute can be used, even though it is not a real database column but a high-level representation of the one-to-many relationship. The `id`

attribute of these new objects is not set explicitly: the primary keys in many databases are managed by the database itself. The objects exist only on the Python side so far; they have not been written to the database yet. Because of that, their `id` values have not yet been assigned:

```
>>> print(admin_role.id)
None
>>> print(mod_role.id)
None
>>> print(user_role.id)
None
```

Changes to the database are managed through a database *session*, which Flask-SQLAlchemy provides as `db.session`. To prepare objects to be written to the database, they must be added to the session:

```
>>> db.session.add(admin_role)
>>> db.session.add(mod_role)
>>> db.session.add(user_role)
>>> db.session.add(user_john)
>>> db.session.add(user_susan)
>>> db.session.add(user_david)
```

Or, more concisely:

```
>>> db.session.add_all([admin_role, mod_role, user_role,
...     user_john, user_susan, user_david])
```

To write the objects to the database, the session needs to be *committed* by calling its `commit()` method:

```
>>> db.session.commit()
```

Check the `id` attributes again after having the data committed to see that they are now set:

```
>>> print(admin_role.id)
1
>>> print(mod_role.id)
2
>>> print(user_role.id)
3
```

> The `db.session` database session is not related to the Flask `session` object discussed in Chapter 4. Database sessions are also called *transactions*.

Database sessions are extremely useful in keeping the database consistent. The commit operation writes all the objects that were added to the session atomically. If an

error occurs while the session is being written, the whole session is discarded. If you always commit related changes together in a session, you are guaranteed to avoid database inconsistencies due to partial updates.

> A database session can also be *rolled back*. If `db.session.rollback()` is called, any objects that were added to the database session are restored to the state they have in the database.

Modifying Rows

The `add()` method of the database session can also be used to update models. Continuing in the same shell session, the following example renames the `"Admin"` role to `"Administrator"`:

```
>>> admin_role.name = 'Administrator'
>>> db.session.add(admin_role)
>>> db.session.commit()
```

Deleting Rows

The database session also has a `delete()` method. The following example deletes the `"Moderator"` role from the database:

```
>>> db.session.delete(mod_role)
>>> db.session.commit()
```

Note that deletions, like insertions and updates, are executed only when the database session is committed.

Querying Rows

Flask-SQLAlchemy makes a `query` object available in each model class. The most basic query for a model is triggered with the `all()` method, which returns the entire contents of the corresponding table:

```
>>> Role.query.all()
[<Role 'Administrator'>, <Role 'User'>]
>>> User.query.all()
[<User 'john'>, <User 'susan'>, <User 'david'>]
```

A query object can be configured to issue more specific database searches through the use of *filters*. The following example finds all the users that were assigned the `"User"` role:

```
>>> User.query.filter_by(role=user_role).all()
[<User 'susan'>, <User 'david'>]
```

It is also possible to inspect the native SQL query that SQLAlchemy generates for a given query by converting the query object to a string:

```
>>> str(User.query.filter_by(role=user_role))
'SELECT users.id AS users_id, users.username AS users_username,
users.role_id AS users_role_id \nFROM users \nWHERE :param_1 = users.role_id'
```

If you exit the shell session, the objects created in the previous example will cease to exist as Python objects but will continue to exist as rows in their respective database tables. If you then start a brand-new shell session, you have to re-create the Python objects from their database rows. The following example issues a query that loads the user role with name "User":

```
>>> user_role = Role.query.filter_by(name='User').first()
```

Note how in this case, the query was issued with the `first()` method instead of `all()`. While `all()` returns all the results of the query as a list, `first()` returns only the first result or `None` if there are no results, so it is a convenient method to use for queries that are known to return one result at the most.

Filters such as `filter_by()` are invoked on a query object and return a new refined query. Multiple filters can be called in sequence until the query is configured as needed.

Table 5-5 shows some of the most common filters available to queries. The complete list is in the SQLAlchemy documentation.

Table 5-5. Common SQLAlchemy query filters

Option	Description
`filter()`	Returns a new query that adds an additional filter to the original query
`filter_by()`	Returns a new query that adds an additional equality filter to the original query
`limit()`	Returns a new query that limits the number of results of the original query to the given number
`offset()`	Returns a new query that applies an offset into the list of results of the original query
`order_by()`	Returns a new query that sorts the results of the original query according to the given criteria
`group_by()`	Returns a new query that groups the results of the original query according to the given criteria

After the desired filters have been applied to the query, a call to `all()` will cause the query to execute and return the results as a list—but there are other ways to trigger the execution of a query besides `all()`. Table 5-6 shows other query execution methods.

Table 5-6. Most common SQLAlchemy query executors

Option	Description
`all()`	Returns all the results of a query as a list
`first()`	Returns the first result of a query, or `None` if there are no results
`first_or_404()`	Returns the first result of a query, or aborts the request and sends a 404 error as the response if there are no results
`get()`	Returns the row that matches the given primary key, or `None` if no matching row is found
`get_or_404()`	Returns the row that matches the given primary key or, if the key is not found, aborts the request and sends a 404 error as the response
`count()`	Returns the result count of the query
`paginate()`	Returns a `Pagination` object that contains the specified range of results

Relationships work similarly to queries. The following example queries the one-to-many relationship between roles and users from both ends:

```
>>> users = user_role.users
>>> users
[<User 'susan'>, <User 'david'>]
>>> users[0].role
<Role 'User'>
```

The `user_role.users` query here has a small problem. The implicit query that runs when the `user_role.users` expression is issued internally calls `all()` to return the list of users. Because the query object is hidden, it is not possible to refine it with additional query filters. In this particular example, it may have been useful to request that the user list be returned in alphabetical order. In Example 5-4, the configuration of the relationship is modified with a `lazy='dynamic'` argument to request that the query is not automatically executed.

Example 5-4. hello.py: dynamic database relationships

```
class Role(db.Model):
    # ...
    users = db.relationship('User', backref='role', lazy='dynamic')
    # ...
```

With the relationship configured in this way, `user_role.users` returns a query that hasn't executed yet, so filters can be added to it:

```
>>> user_role.users.order_by(User.username).all()
[<User 'david'>, <User 'susan'>]
>>> user_role.users.count()
2
```

Database Use in View Functions

The database operations described in the previous sections can be used directly inside view functions. Example 5-5 shows a new version of the home page route that records names entered by users in the database.

Example 5-5. hello.py: database use in view functions

```
@app.route('/', methods=['GET', 'POST'])
def index():
    form = NameForm()
    if form.validate_on_submit():
        user = User.query.filter_by(username=form.name.data).first()
        if user is None:
            user = User(username=form.name.data)
            db.session.add(user)
            db.session.commit()
            session['known'] = False
        else:
            session['known'] = True
        session['name'] = form.name.data
        form.name.data = ''
        return redirect(url_for('index'))
    return render_template('index.html',
        form=form, name=session.get('name'),
        known=session.get('known', False))
```

In this modified version of the application, each time a name is submitted the application checks for it in the database using the `filter_by()` query filter. A `known` variable is written to the user session so that after the redirect the information can be sent to the template, where it is used to customize the greeting. Note that for the application to work, the database tables must be created in a Python shell as shown earlier.

The new version of the associated template is shown in Example 5-6. This template uses the `known` argument to add a second line to the greeting that is different for known and new users.

Example 5-6. templates/index.html: customized greeting in template

```
{% extends "base.html" %}
{% import "bootstrap/wtf.html" as wtf %}

{% block title %}Flasky{% endblock %}

{% block page_content %}
<div class="page-header">
    <h1>Hello, {% if name %}{{ name }}{% else %}Stranger{% endif %}!</h1>
    {% if not known %}
```

```
    <p>Pleased to meet you!</p>
    {% else %}
    <p>Happy to see you again!</p>
    {% endif %}
</div>
{{ wtf.quick_form(form) }}
{% endblock %}
```

If you have cloned the application's Git repository on GitHub, you can run `git checkout 5b` to check out this version of the application.

Integration with the Python Shell

Having to import the database instance and the models each time a shell session is started is tedious work. To avoid having to constantly repeat these steps, the `flask shell` command can be configured to automatically import these objects.

To add objects to the import list, a *shell context processor* must be created and registered with the `app.shell_context_processor` decorator. This is shown in Example 5-7.

Example 5-7. hello.py: adding a shell context

```python
@app.shell_context_processor
def make_shell_context():
    return dict(db=db, User=User, Role=Role)
```

The shell context processor function returns a dictionary that includes the database instance and the models. The `flask shell` command will import these items automatically into the shell, in addition to `app`, which is imported by default:

```
$ flask shell
>>> app
<Flask 'hello'>
>>> db
<SQLAlchemy engine='sqlite:////home/flask/flasky/data.sqlite'>
>>> User
<class 'hello.User'>
```

If you have cloned the application's Git repository on GitHub, you can run `git checkout 5c` to check out this version of the application.

Database Migrations with Flask-Migrate

As you make progress developing an application, you will find that your database models need to change, and when that happens the database needs to be updated as well. Flask-SQLAlchemy creates database tables from models only when they do not exist already, so the only way to make it update tables is by destroying the old tables first—but of course, this causes all the data in the database to be lost.

A better solution is to use a *database migration* framework. In the same way source code version control tools keep track of changes to source code files, a database migration framework keeps track of changes to a database *schema*, allowing incremental changes to be applied.

The developer of SQLAlchemy has written a migration framework called Alembic, but instead of using Alembic directly, Flask applications can use the Flask-Migrate extension, a lightweight Alembic wrapper that integrates it with the `flask` command.

Creating a Migration Repository

To begin, Flask-Migrate must be installed in the virtual environment:

```
(venv) $ pip install flask-migrate
```

Example 5-8 shows how the extension is initialized.

Example 5-8. hello.py: Flask-Migrate initialization

```
from flask_migrate import Migrate

# ...

migrate = Migrate(app, db)
```

To expose the database migration commands, Flask-Migrate adds a `flask db` command with several subcommands. When you work on a new project, you can add support for database migrations with the `init` subcommand:

```
(venv) $ flask db init
  Creating directory /home/flask/flasky/migrations...done
  Creating directory /home/flask/flasky/migrations/versions...done
  Generating /home/flask/flasky/migrations/alembic.ini...done
  Generating /home/flask/flasky/migrations/env.py...done
  Generating /home/flask/flasky/migrations/env.pyc...done
  Generating /home/flask/flasky/migrations/README...done
  Generating /home/flask/flasky/migrations/script.py.mako...done
  Please edit configuration/connection/logging settings in
  '/home/flask/flasky/migrations/alembic.ini' before proceeding.
```

This command creates a *migrations* directory, where all the migration scripts will be stored. If you are following the example project using `git checkout`, you do not need to do this step, as the migration repository is already included in the GitHub repository.

> The files in a database migration repository must always be added to version control along with the rest of the application.

Creating a Migration Script

In Alembic, a database migration is represented by a *migration script*. This script has two functions called `upgrade()` and `downgrade()`. The `upgrade()` function applies the database changes that are part of the migration, and the `downgrade()` function removes them. This ability to add and remove changes means, Alembic can reconfigure a database to any point in the change history.

Alembic migrations can be created manually or automatically using the `revision` and `migrate` commands, respectively. A manual migration creates a migration skeleton script with empty `upgrade()` and `downgrade()` functions that need to be implemented by the developer using directives exposed by Alembic's `Operations` object. An automatic migration attempts to generate the code for the `upgrade()` and `downgrade()` functions by looking for differences between the model definitions and the current state of the database.

> Automatic migrations are not always accurate and can miss some details that are ambiguous. For example, if a column is renamed, an automatically generated migration may show that the column in question was deleted and a new column was added with the new name. Leaving the migration as is will cause the data in this column to be lost! For this reason, migration scripts generated automatically should always be reviewed and manually corrected if they have any inaccuracies.

To make changes to your database schema with Flask-Migrate, the following procedure needs to be followed:

1. Make the necessary changes to the model classes.
2. Create an automatic migration script with the `flask db migrate` command.
3. Review the generated script and adjust it so that it accurately represents the changes that were made to the models.

4. Add the migration script to source control.
5. Apply the migration to the database with the `flask db upgrade` command.

The `flask db migrate` subcommand creates an automatic migration script:

```
(venv) $ flask db migrate -m "initial migration"
INFO  [alembic.migration] Context impl SQLiteImpl.
INFO  [alembic.migration] Will assume non-transactional DDL.
INFO  [alembic.autogenerate] Detected added table 'roles'
INFO  [alembic.autogenerate] Detected added table 'users'
INFO  [alembic.autogenerate.compare] Detected added index
'ix_users_username' on '['username']'
  Generating /home/flask/flasky/migrations/versions/1bc
  594146bb5_initial_migration.py...done
```

If you are following the `git checkout` instructions to incrementally update the example application, you do not need to issue the `migrate` commands, as the migration scripts are already incorporated into the Git repository tags.

> If you have cloned the application's Git repository on GitHub, you can run `git checkout 5d` to check out this version of the application. Note that you do not need to generate the migration repository and the migration scripts for this application as these are included in the GitHub repository.

Upgrading the Database

Once a migration script has been reviewed and accepted, it can be applied to the database using the `flask db upgrade` command:

```
(venv) $ flask db upgrade
INFO  [alembic.migration] Context impl SQLiteImpl.
INFO  [alembic.migration] Will assume non-transactional DDL.
INFO  [alembic.migration] Running upgrade None -> 1bc594146bb5, initial migration
```

For a first migration, this is effectively equivalent to calling `db.create_all()`, but in successive migrations the `flask db upgrade` command applies updates to the tables without affecting their contents.

If you have been working with the application in its previous stages, you already have a database file that was created with the `db.create_all()` function earlier. In this state, the `flask db upgrade` will fail because it will try to create database tables that already exist. A simple way to address this problem is to delete your *data.sqlite* database file and then run `flask db upgrade` to generate a new database through the migration framework. Another option is to skip the `flask db upgrade` and instead mark the existing database as upgraded using the `flask db stamp` command.

Adding More Migrations

As you work on your own projects, you are going to find that you need to make changes to your database models very often. When you manage the database through a migration framework, all changes must be defined in migration scripts, because anything that is not tracked in a migration will not be repeatable. The procedure to introduce a change in the database is similar to what was done to introduce the first migration:

1. Make the necessary changes in the database models.
2. Generate a migration with the `flask db migrate` command.
3. Review the generated migration script and correct it if it has any inaccuracies.
4. Apply the changes to the database with the `flask db upgrade` command.

While working on a specific feature, you may find that you need to make several changes to your database models before you get them the way you want them. If your last migration has not been committed to source control yet, you can opt to expand it to incorporate new changes as you make them, and this will save you from having lots of very small migration scripts that are meaningless on their own. The procedure to expand the last migration script is as follows:

1. Remove the last migration from the database with the `flask db downgrade` command (note that this may cause some data to be lost).
2. Delete the last migration script, which is now orphaned.
3. Generate a new database migration with the `flask db migrate` command, which will now include the changes in the migration script you just removed, plus any other changes you've made to the models.
4. Review and apply the migration script as described previously.

Consult the Flask-Migrate documentation to learn about other subcommands related to database migrations.

The topic of database design and usage is very important; entire books have been written on the subject. You should consider this chapter as an overview; more advanced topics will be discussed in later chapters. The next chapter is dedicated to sending email.

CHAPTER 6

Email

Many types of applications need to notify users when certain events occur, and the usual method of communication is email. In this chapter you are going to learn how to send emails from a Flask application.

Email Support with Flask-Mail

Although the `smtplib` package from the Python standard library can be used to send email inside a Flask application, the Flask-Mail extension wraps `smtplib` and integrates it nicely with Flask. Flask-Mail is installed with *pip*:

```
(venv) $ pip install flask-mail
```

The extension connects to a Simple Mail Transfer Protocol (SMTP) server and passes emails to it for delivery. If no configuration is given, Flask-Mail connects to *localhost* at port 25 and sends email without authentication. Table 6-1 shows the list of configuration keys that can be used to configure the SMTP server.

Table 6-1. Flask-Mail SMTP server configuration keys

Key	Default	Description
`MAIL_SERVER`	*localhost*	Hostname or IP address of the email server
`MAIL_PORT`	25	Port of the email server
`MAIL_USE_TLS`	`False`	Enable Transport Layer Security (TLS) security
`MAIL_USE_SSL`	`False`	Enable Secure Sockets Layer (SSL) security
`MAIL_USERNAME`	`None`	Mail account username
`MAIL_PASSWORD`	`None`	Mail account password

During development it may be more convenient to connect to an external SMTP server. As an example, Example 6-1 shows how to configure the application to send email through a Google Gmail account.

Example 6-1. hello.py: Flask-Mail configuration for Gmail

```
import os
# ...
app.config['MAIL_SERVER'] = 'smtp.googlemail.com'
app.config['MAIL_PORT'] = 587
app.config['MAIL_USE_TLS'] = True
app.config['MAIL_USERNAME'] = os.environ.get('MAIL_USERNAME')
app.config['MAIL_PASSWORD'] = os.environ.get('MAIL_PASSWORD')
```

> Never write account credentials directly in your scripts, particularly if you plan to release your work as open source. To protect your account information, have your script import sensitive information from environment variables.

> For security reasons, Gmail accounts are configured to require external applications to use OAuth2 authentication to connect to the email server. Unfortunately, Python's `smtplib` library does not support this method of authentication. To make your Gmail account accept standard SMTP authentication, go to your Google account settings page and select "Signing in to Google" from the left menu bar. On that page, locate the "Allow less secure apps" setting and make sure it is enabled. If enabling this setting on your personal Gmail account concerns you, create a secondary account only to test sending emails.

Flask-Mail is initialized as shown in Example 6-2.

Example 6-2. hello.py: Flask-Mail initialization

```
from flask_mail import Mail
mail = Mail(app)
```

The two environment variables that hold the email server username and password need to be defined in the environment. If you are on Linux or macOS, you can set these variables as follows:

```
(venv) $ export MAIL_USERNAME=<Gmail username>
(venv) $ export MAIL_PASSWORD=<Gmail password>
```

For Microsoft Windows users, the environment variables are set as follows:

```
(venv) $ set MAIL_USERNAME=<Gmail username>
(venv) $ set MAIL_PASSWORD=<Gmail password>
```

Sending Email from the Python Shell

To test the configuration, you can start a shell session and send a test email (replace `you@example.com` with your own email address):

```
(venv) $ flask shell
>>> from flask_mail import Message
>>> from hello import mail
>>> msg = Message('test email', sender='you@example.com',
...     recipients=['you@example.com'])
>>> msg.body = 'This is the plain text body'
>>> msg.html = 'This is the <b>HTML</b> body'
>>> with app.app_context():
...    mail.send(msg)
...
```

Note that Flask-Mail's `send()` function uses `current_app`, so it needs to be executed with an activated application context.

Integrating Emails with the Application

To avoid having to create email messages manually every time, it is a good idea to abstract the common parts of the application's email sending functionality into a function. As an added benefit, this function can render email bodies from Jinja2 templates to have the most flexibility. The implementation is shown in Example 6-3.

Example 6-3. hello.py: email support

```python
from flask_mail import Message

app.config['FLASKY_MAIL_SUBJECT_PREFIX'] = '[Flasky]'
app.config['FLASKY_MAIL_SENDER'] = 'Flasky Admin <flasky@example.com>'

def send_email(to, subject, template, **kwargs):
    msg = Message(app.config['FLASKY_MAIL_SUBJECT_PREFIX'] + subject,
                  sender=app.config['FLASKY_MAIL_SENDER'], recipients=[to])
    msg.body = render_template(template + '.txt', **kwargs)
    msg.html = render_template(template + '.html', **kwargs)
    mail.send(msg)
```

The function relies on two application-specific configuration keys that define a prefix string for the subject and the address that will be used as the sender. The `send_email()` function takes the destination address, a subject line, a template for the email body, and a list of keyword arguments. The template name must be given

without the extension, so that two versions of the template can be used for the plain text and HTML bodies. The keyword arguments passed by the caller are given to `render_template()` so that they can be used by the templates that generate the email body as template variables.

The `index()` view function can easily be expanded to send an email to the administrator whenever a new name is received with the form. Example 6-4 shows this change.

Example 6-4. hello.py: email example

```
# ...
app.config['FLASKY_ADMIN'] = os.environ.get('FLASKY_ADMIN')
# ...
@app.route('/', methods=['GET', 'POST'])
def index():
    form = NameForm()
    if form.validate_on_submit():
        user = User.query.filter_by(username=form.name.data).first()
        if user is None:
            user = User(username=form.name.data)
            db.session.add(user)
            session['known'] = False
            if app.config['FLASKY_ADMIN']:
                send_email(app.config['FLASKY_ADMIN'], 'New User',
                           'mail/new_user', user=user)
        else:
            session['known'] = True
        session['name'] = form.name.data
        form.name.data = ''
        return redirect(url_for('index'))
    return render_template('index.html', form=form, name=session.get('name'),
                           known=session.get('known', False))
```

The recipient of the email is given in the `FLASKY_ADMIN` environment variable that's loaded into a configuration variable of the same name during startup. Two template files need to be created for the text and HTML versions of the email. These files are stored in a *mail* subdirectory inside *templates* to keep them separate from regular templates. The email templates expect the user to be given as a template argument, so the call to `send_email()` includes it as a keyword argument.

> If you have cloned the application's Git repository on GitHub, you can run `git checkout 6a` to check out this version of the application.

In addition to the MAIL_USERNAME and MAIL_PASSWORD environment variables described earlier, this version of the application needs the FLASKY_ADMIN environment variable. For Linux and macOS users, the command to set this variable is:

```
(venv) $ export FLASKY_ADMIN=<your-email-address>
```

For Microsoft Windows users, this is the equivalent command:

```
(venv) $ set FLASKY_ADMIN=<your-email-address>
```

With these environment variables set, you can test the application and receive an email every time you enter a new name in the form.

Sending Asynchronous Email

If you sent a few test emails, you likely noticed that the mail.send() function blocks for a few seconds while the email is sent, making the browser look unresponsive during that time. To avoid unnecessary delays during request handling, the email send function can be moved to a background thread. Example 6-5 shows this change.

Example 6-5. hello.py: asynchronous email support

```python
from threading import Thread

def send_async_email(app, msg):
    with app.app_context():
        mail.send(msg)

def send_email(to, subject, template, **kwargs):
    msg = Message(app.config['FLASKY_MAIL_SUBJECT_PREFIX'] + subject,
                  sender=app.config['FLASKY_MAIL_SENDER'], recipients=[to])
    msg.body = render_template(template + '.txt', **kwargs)
    msg.html = render_template(template + '.html', **kwargs)
    thr = Thread(target=send_async_email, args=[app, msg])
    thr.start()
    return thr
```

This implementation highlights an interesting problem. Many Flask extensions operate under the assumption that there are active application and/or request contexts. As mentioned previously, Flask-Mail's send() function uses current_app, so it requires the application context to be active. But since contexts are associated with a thread, when the mail.send() function executes in a different thread it needs the application context to be created artificially using app.app_context(). The app instance is passed to the thread as an argument so that a context can be created.

If you have cloned the application's Git repository on GitHub, you can run `git checkout 6b` to check out this version of the application.

If you run the application now, you will notice that it is much more responsive, but keep in mind that for applications that send a large volume of email, having a job dedicated to sending email is more appropriate than starting a new thread for each email send operation. For example, the execution of the `send_async_email()` function can be sent to a Celery task queue.

This chapter completes the overview of the features that are a must-have for most web applications. The problem now is that the *hello.py* script is starting to get large, and that makes it harder to work with. In the next chapter, you will learn how to structure a larger application.

CHAPTER 7

Large Application Structure

Although having small web applications stored in a single script file can be very convenient, this approach does not scale well. As the application grows in complexity, working with a single large source file becomes problematic.

Unlike most other web frameworks, Flask does not impose a specific organization for large projects; the way to structure the application is left entirely to the developer. In this chapter, a possible way to organize a large application in packages and modules is presented. This structure will be used in the remaining examples of the book.

Project Structure

Example 7-1 shows the basic layout for a Flask application.

Example 7-1. Basic multiple-file Flask application structure

```
|-flasky
  |-app/
    |-templates/
    |-static/
    |-main/
      |-__init__.py
      |-errors.py
      |-forms.py
      |-views.py
    |-__init__.py
    |-email.py
    |-models.py
  |-migrations/
  |-tests/
    |-__init__.py
    |-test*.py
  |-venv/
  |-requirements.txt
  |-config.py
  |-flasky.py
```

This structure has four top-level folders:

- The Flask application lives inside a package generically named *app*.
- The *migrations* folder contains the database migration scripts, as before.
- Unit tests are written in a *tests* package.
- The *venv* folder contains the Python virtual environment, as before.

There are also a few new files:

- *requirements.txt* lists the package dependencies so that it is easy to regenerate an identical virtual environment on a different computer.
- *config.py* stores the configuration settings.
- *flasky.py* defines the Flask application instance, and also includes a few tasks that help manage the application.

To help you fully understand this structure, the following sections describe the process to convert the *hello.py* application to it.

Configuration Options

Applications often need several configuration sets. The best example of this is the need to use different databases during development, testing, and production so that they don't interfere with each other.

Instead of the simple `app.config` dictionary-like configuration used by *hello.py*, a hierarchy of configuration classes can be used. Example 7-2 shows the *config.py* file, with all the settings imported from *hello.py*.

Example 7-2. config.py: application configuration

```
import os
basedir = os.path.abspath(os.path.dirname(__file__))

class Config:
    SECRET_KEY = os.environ.get('SECRET_KEY') or 'hard to guess string'
    MAIL_SERVER = os.environ.get('MAIL_SERVER', 'smtp.googlemail.com')
    MAIL_PORT = int(os.environ.get('MAIL_PORT', '587'))
    MAIL_USE_TLS = os.environ.get('MAIL_USE_TLS', 'true').lower() in \
        ['true', 'on', '1']
    MAIL_USERNAME = os.environ.get('MAIL_USERNAME')
    MAIL_PASSWORD = os.environ.get('MAIL_PASSWORD')
    FLASKY_MAIL_SUBJECT_PREFIX = '[Flasky]'
    FLASKY_MAIL_SENDER = 'Flasky Admin <flasky@example.com>'
    FLASKY_ADMIN = os.environ.get('FLASKY_ADMIN')
    SQLALCHEMY_TRACK_MODIFICATIONS = False
```

```
    @staticmethod
    def init_app(app):
        pass

class DevelopmentConfig(Config):
    DEBUG = True
    SQLALCHEMY_DATABASE_URI = os.environ.get('DEV_DATABASE_URL') or \
        'sqlite:///' + os.path.join(basedir, 'data-dev.sqlite')

class TestingConfig(Config):
    TESTING = True
    SQLALCHEMY_DATABASE_URI = os.environ.get('TEST_DATABASE_URL') or \
        'sqlite://'

class ProductionConfig(Config):
    SQLALCHEMY_DATABASE_URI = os.environ.get('DATABASE_URL') or \
        'sqlite:///' + os.path.join(basedir, 'data.sqlite')

config = {
    'development': DevelopmentConfig,
    'testing': TestingConfig,
    'production': ProductionConfig,

    'default': DevelopmentConfig
}
```

The `Config` base class contains settings that are common to all configurations; the different subclasses define settings that are specific to a configuration. Additional configurations can be added as needed.

To make configuration more flexible and safe, most settings can be optionally imported from environment variables. For example, the value of the `SECRET_KEY`, due to its sensitive nature, can be set in the environment, but a default value is provided in case the environment does not define it. Typically, these settings can be used with their defaults during development but should each have an appropriate value set in the corresponding environment variable on the production server. The configuration options for the email server are all imported from environment variables as well, with defaults pointing to the Gmail server for convenience during development.

> Never write passwords or other secrets in a configuration file that is committed to source control.

The `SQLALCHEMY_DATABASE_URI` variable is assigned different values under each of the three configurations. This enables the application to use a different database in each

configuration. This is very important, as you don't want a run of the unit tests to change the database that you use for day-to-day development. Each configuration tries to import the database URL from an environment variable, and when that is not available it sets a default one based on SQLite. For the testing configuration, the default is an in-memory database, since there is no need to store any data outside of the test run.

The development and production configurations each have a set of mail server configuration options. As an additional way to allow the application to customize its configuration, the `Config` class and its subclasses can define an `init_app()` class method that takes the application instance as an argument. For now the base `Config` class implements an empty `init_app()` method.

At the bottom of the configuration script, the different configurations are registered in a `config` dictionary. One of the configurations (the one for development, in this case) is also registered as the default.

Application Package

The application package is where all the application code, templates, and static files live. It is called simply *app* here, though it can be given an application-specific name if desired. The *templates* and *static* directories are now part of the application package, so they are moved inside *app*. The database models and the email support functions are also moved inside this package, each in its own module, as *app/models.py* and *app/email.py*.

Using an Application Factory

The way the application is created in the single-file version is very convenient, but it has one big drawback. Because the application is created in the global scope, there is no way to apply configuration changes dynamically: by the time the script is running, the application instance has already been created, so it is already too late to make configuration changes. This is particularly important for unit tests because sometimes it is necessary to run the application under different configuration settings for better test coverage.

The solution to this problem is to delay the creation of the application by moving it into a *factory function* that can be explicitly invoked from the script. This not only gives the script time to set the configuration, but also the ability to create multiple application instances—another thing that can be very useful during testing. The application factory function, shown in Example 7-3, is defined in the *app* package constructor.

Example 7-3. app/__init__.py: application package constructor

```
from flask import Flask, render_template
from flask_bootstrap import Bootstrap
from flask_mail import Mail
from flask_moment import Moment
from flask_sqlalchemy import SQLAlchemy
from config import config

bootstrap = Bootstrap()
mail = Mail()
moment = Moment()
db = SQLAlchemy()

def create_app(config_name):
    app = Flask(__name__)
    app.config.from_object(config[config_name])
    config[config_name].init_app(app)

    bootstrap.init_app(app)
    mail.init_app(app)
    moment.init_app(app)
    db.init_app(app)

    # attach routes and custom error pages here

    return app
```

This constructor imports most of the Flask extensions currently in use, but because there is no application instance to initialize them with, it creates them uninitialized by passing no arguments into their constructors. The `create_app()` function is the application factory, which takes as an argument the name of a configuration to use for the application. The configuration settings stored in one of the classes defined in *config.py* can be imported directly into the application using the `from_object()` method available in Flask's `app.config` configuration object. The configuration object is selected by name from the `config` dictionary. Once an application is created and configured, the extensions can be initialized. Calling `init_app()` on the extensions that were created earlier completes their initialization.

The application initialization is now done in this factory function, using the `from_object()` method from the Flask configuration object, which takes as an argument one of the configuration classes defined in *config.py*. The `init_app()` method of the selected configuration is also invoked, to allow more complex initialization procedures to take place.

The factory function returns the created application instance, but note that applications created with the factory function in its current state are incomplete, as they are missing routes and custom error page handlers. This is the topic of the next section.

Implementing Application Functionality in a Blueprint

The conversion to an application factory introduces a complication for routes. In single-script applications, the application instance exists in the global scope, so routes can be easily defined using the `app.route` decorator. But now that the application is created at runtime, the `app.route` decorator begins to exist only after `create_app()` is invoked, which is too late. Custom error page handlers present the same problem, as these are defined with the `app.errorhandler` decorator.

Luckily, Flask offers a better solution using *blueprints*. A blueprint is similar to an application in that it can also define routes and error handlers. The difference is that when these are defined in a blueprint they are in a dormant state until the blueprint is registered with an application, at which point they become part of it. Using a blueprint defined in the global scope, the routes and error handlers of the application can be defined in almost the same way as in the single-script application.

Like applications, blueprints can be defined all in a single file or can be created in a more structured way with multiple modules inside a package. To allow for the greatest flexibility, a subpackage inside the application package will be created to host the first blueprint of the application. Example 7-4 shows the package constructor, which creates the blueprint.

Example 7-4. app/main/__init__.py: main blueprint creation

```
from flask import Blueprint

main = Blueprint('main', __name__)

from . import views, errors
```

Blueprints are created by instantiating an object of class `Blueprint`. The constructor for this class takes two required arguments: the blueprint name and the module or package where the blueprint is located. As with applications, Python's `__name__` variable is in most cases the correct value for the second argument.

The routes of the application are stored in an *app/main/views.py* module inside the package, and the error handlers are in *app/main/errors.py*. Importing these modules causes the routes and error handlers to be associated with the blueprint. It is important to note that the modules are imported at the bottom of the *app/main/__init__.py* script to avoid errors due to circular dependencies. In this particular example the problem is that *app/main/views.py* and *app/main/errors.py* in turn are going to import the `main` blueprint object, so the imports are going to fail unless the circular reference occurs after `main` is defined.

> The from . import <some-module> syntax is used in Python to represent *relative imports*. The . in this statement represents the current package. You are going to see another very useful relative import soon that uses the form from .. import <some-module>, where .. represents the parent of the current package.

The blueprint is registered with the application inside the `create_app()` factory function, as shown in Example 7-5.

Example 7-5. app/__init__.py: main blueprint registration

```
def create_app(config_name):
    # ...

    from .main import main as main_blueprint
    app.register_blueprint(main_blueprint)

    return app
```

Example 7-6 shows the error handlers.

Example 7-6. app/main/errors.py: error handlers in main blueprint

```
from flask import render_template
from . import main

@main.app_errorhandler(404)
def page_not_found(e):
    return render_template('404.html'), 404

@main.app_errorhandler(500)
def internal_server_error(e):
    return render_template('500.html'), 500
```

A difference when writing error handlers inside a blueprint is that if the `errorhandler` decorator is used, the handler will be invoked only for errors that originate in the routes defined by the blueprint. To install application-wide error handlers, the `app_errorhandler` decorator must be used instead.

Example 7-7 shows the route of the application updated to be in the blueprint.

Example 7-7. app/main/views.py: application routes in main blueprint

```
from datetime import datetime
from flask import render_template, session, redirect, url_for
from . import main
from .forms import NameForm
from .. import db
from ..models import User

@main.route('/', methods=['GET', 'POST'])
def index():
    form = NameForm()
    if form.validate_on_submit():
        # ...
        return redirect(url_for('.index'))
    return render_template('index.html',
                           form=form, name=session.get('name'),
                           known=session.get('known', False),
                           current_time=datetime.utcnow())
```

There are two main differences when writing a view function inside a blueprint. First, as was done for error handlers earlier, the route decorator comes from the blueprint, so `main.route` is used instead of `app.route`. The second difference is in the usage of the `url_for()` function. As you may recall, the first argument to this function is the endpoint name of the route, which for application-based routes defaults to the name of the view function. For example, in a single-script application the URL for an `index()` view function can be obtained with `url_for('index')`.

The difference with blueprints is that Flask applies a namespace to all the endpoints defined in a blueprint, so that multiple blueprints can define view functions with the same endpoint names without collisions. The namespace is the name of the blueprint (the first argument to the `Blueprint` constructor) and is separated from the endpoint name with a dot. The `index()` view function is then registered with endpoint name `main.index` and its URL can be obtained with `url_for('main.index')`.

The `url_for()` function also supports a shorter format for endpoints in blueprints in which the blueprint name is omitted, such as `url_for('.index')`. With this notation, the blueprint name for the current request is used to complete the endpoint name. This effectively means that redirects within the same blueprint can use the shorter form, while redirects across blueprints must use the fully qualified endpoint name that includes the blueprint name.

To complete the changes to the application package, the form objects are also stored inside the blueprint in the *app/main/forms.py* module.

Application Script

The *flasky.py* module in the top-level directory is where the application instance is defined. This script is shown in Example 7-8.

Example 7-8. flasky.py: main script

```
import os
from app import create_app, db
from app.models import User, Role
from flask_migrate import Migrate

app = create_app(os.getenv('FLASK_CONFIG') or 'default')
migrate = Migrate(app, db)

@app.shell_context_processor
def make_shell_context():
    return dict(db=db, User=User, Role=Role)
```

The script begins by creating an application. The configuration is taken from the environment variable `FLASK_CONFIG` if it's defined, or else the default configuration is used. Flask-Migrate and the custom context for the Python shell are then initialized.

Because the main script of the application changed from *hello.py* to *flasky.py*, the `FLASK_APP` environment variable needs to be updated accordingly so that the `flask` command can locate the application instance. It is also useful to enable Flask's debug mode by setting `FLASK_DEBUG=1`. For Linux and macOS, this is all done as follows:

```
(venv) $ export FLASK_APP=flasky.py
(venv) $ export FLASK_DEBUG=1
```

And for Microsoft Windows:

```
(venv) $ set FLASK_APP=flasky.py
(venv) $ set FLASK_DEBUG=1
```

Requirements File

It is a good practice for applications to include a *requirements.txt* file that records all the package dependencies, with the exact version numbers. This is important in case the virtual environment needs to be regenerated on a different machine, such as the machine on which the application will be deployed for production use. This file can be generated automatically by *pip* with the following command:

```
(venv) $ pip freeze >requirements.txt
```

It is a good idea to refresh this file whenever a package is installed or upgraded. An example requirements file is shown here:

```
alembic==0.9.3
blinker==1.4
click==6.7
dominate==2.3.1
Flask==0.12.2
Flask-Bootstrap==3.3.7.1
Flask-Mail==0.9.1
Flask-Migrate==2.0.4
Flask-Moment==0.5.1
Flask-SQLAlchemy==2.2
Flask-WTF==0.14.2
itsdangerous==0.24
Jinja2==2.9.6
Mako==1.0.7
MarkupSafe==1.0
python-dateutil==2.6.1
python-editor==1.0.3
six==1.10.0
SQLAlchemy==1.1.11
visitor==0.1.3
Werkzeug==0.12.2
WTForms==2.1
```

When you need to build a perfect replica of the virtual environment, you can create a new virtual environment and run the following command on it:

```
(venv) $ pip install -r requirements.txt
```

The version numbers in the example *requirements.txt* file are likely going to be outdated by the time you read this. You can try using more recent releases of the packages, if you like. If you experience any problems, you can always go back to the versions specified here, as those are known to be compatible with the application.

Unit Tests

This application is very small, so there isn't a lot to test yet. But as an example, two simple tests can be defined, as shown in Example 7-9.

Example 7-9. tests/test_basics.py: unit tests

```python
import unittest
from flask import current_app
from app import create_app, db

class BasicsTestCase(unittest.TestCase):
    def setUp(self):
        self.app = create_app('testing')
        self.app_context = self.app.app_context()
        self.app_context.push()
        db.create_all()
```

```
    def tearDown(self):
        db.session.remove()
        db.drop_all()
        self.app_context.pop()

    def test_app_exists(self):
        self.assertFalse(current_app is None)

    def test_app_is_testing(self):
        self.assertTrue(current_app.config['TESTING'])
```

The tests are written using the standard `unittest` package from the Python standard library. The `setUp()` and `tearDown()` methods of the test case class run before and after each test, and any methods that have a name that begins with `test_` are executed as tests.

> If you want to learn more about writing unit tests with Python's `unittest` package, read the official documentation.

The `setUp()` method tries to create an environment for the test that is close to that of a running application. It first creates an application configured for testing and activates its context. This step ensures that tests have access to `current_app`, like regular requests do. Then it creates a brand-new database for the tests using Flask-SQLAlchemy's `create_all()` method. The database and the application context are removed in the `tearDown()` method.

The first test ensures that the application instance exists. The second test ensures that the application is running under the testing configuration. To make the *tests* directory a proper package, a *tests/init.py* module needs to be added, but this can be an empty file, as the `unittest` package scans all the modules to discover the tests.

> If you have cloned the application's Git repository on GitHub, you can run `git checkout 7a` to check out the converted version of the application. To ensure that you have all the dependencies installed, also run `pip install -r requirements.txt`.

To run the unit tests, a custom command can be added to the *flasky.py* script. Example 7-10 shows how to add a `test` command.

Example 7-10. flasky.py: unit test launcher command

```
@app.cli.command()
def test():
    """Run the unit tests."""
    import unittest
    tests = unittest.TestLoader().discover('tests')
    unittest.TextTestRunner(verbosity=2).run(tests)
```

The `app.cli.command` decorator makes it simple to implement custom commands. The name of the decorated function is used as the command name, and the function's docstring is displayed in the help messages. The implementation of the `test()` function invokes the test runner from the `unittest` package.

The unit tests can be executed as follows:

```
(venv) $ flask test
test_app_exists (test_basics.BasicsTestCase) ... ok
test_app_is_testing (test_basics.BasicsTestCase) ... ok

.----------------------------------------------------------------------
Ran 2 tests in 0.001s

OK
----
```

Database Setup

The restructured application uses a different database than the single-script version.

The database URL is taken from an environment variable as a first choice, with a default SQLite database as an alternative. The environment variables and SQLite database filenames are different for each of the three configurations. For example, in the development configuration the URL is obtained from the environment variable `DEV_DATABASE_URL`, and if that is not defined then an SQLite database with the name *data-dev.sqlite* is used.

Regardless of the source of the database URL, the database tables must be created for the new database. When working with Flask-Migrate to keep track of migrations, database tables can be created or upgraded to the latest revision with a single command:

```
(venv) $ flask db upgrade
```

Running the Application

The refactoring is now complete, and the application can be started. Make sure you have updated the `FLASK_APP` environment variable as indicated in "Application Script" on page 93, and then run the application as usual:

```
(venv) $ flask run
```

Having to set the `FLASK_APP` and `FLASK_DEBUG` environment variables every time a new command-prompt session is started can get tedious, so you should configure your system so that these variables are set by default. If you are using *bash*, you can add them to your *~/.bashrc* file.

Believe it or not, you have reached the end of Part I. You have now learned about the basic elements necessary to build a web application with Flask, but you probably feel unsure about how all these pieces fit together to form a real application. The goal of Part II is to help with that by walking you through the development of a complete application.

PART II

Example: A Social Blogging Application

CHAPTER 8

User Authentication

Most applications need to keep track of who their users are. When users connect with an application, they *authenticate* with it, a process by which they make their identity known. Once the application knows who the user is, it can offer a customized experience.

The most commonly used method of authentication requires users to provide a piece of identification, which is either their email address or username, and a secret only known to them, which is called the password. In this chapter, the complete authentication system for Flasky is created.

Authentication Extensions for Flask

There are many excellent Python authentication packages, but none of them do everything. The user authentication solution presented in this chapter uses several packages and provides the glue that makes them work well together. This is the list of packages that will be used, and what they're used for:

- Flask-Login: Management of user sessions for logged-in users
- Werkzeug: Password hashing and verification
- `itsdangerous`: Cryptographically secure token generation and verification

In addition to authentication-specific packages, the following general-purpose extensions will be used:

- Flask-Mail: Sending of authentication-related emails
- Flask-Bootstrap: HTML templates
- Flask-WTF: Web forms

Password Security

The safety of user information stored in databases is often overlooked during the design of web applications. If an attacker is able to break into your server and access your user database, then you risk the security of your users—and the risk is bigger than you think. It is a known fact that most users use the same password on multiple sites, so even if you don't store any sensitive information, access to the passwords stored in your database can give the attacker access to accounts your users have on other sites.

The key to storing user passwords securely in a database relies on not storing the password itself but a *hash* of it. A password hashing function takes a password as input, adds a random component to it (the *salt*), and then applies several one-way cryptographic transformations to it. The result is a new sequence of characters that has no resemblance to the original password, and has no known way to be transformed back into the original password. Password hashes can be verified in place of the real passwords because hashing functions are repeatable: given the same inputs (the password and the salt), the result is always the same.

> Password hashing is a complex task that is hard to get right. It is recommended that you don't implement your own solution but instead rely on well-known libraries that have been reviewed by the community. In the next section, Werkzeug's password hashing functions will be demonstrated. Other good choices for password hashing are bcrypt and Passlib. If you are interested in learning what's involved in generating secure password hashes, the article "Salted Password Hashing - Doing It Right" by Defuse Security is a worthwhile read.

Hashing Passwords with Werkzeug

Werkzeug's *security* module conveniently implements secure password hashing. This functionality is exposed with just two functions, used in the registration and verification phases, respectively:

`generate_password_hash(password, method='pbkdf2:sha256', salt_length=8)`
: This function takes a plain-text password and returns the password hash as a string that can be stored in the user database. The default values for `method` and `salt_length` are sufficient for most use cases.

`check_password_hash(hash, password)`
: This function takes a password hash previously stored in the database and the password entered by the user. A return value of `True` indicates that the user password is correct.

Example 8-1 shows the changes to the User model created in Chapter 5 to accommodate password hashing.

Example 8-1. app/models.py: password hashing in the User model

```
from werkzeug.security import generate_password_hash, check_password_hash

class User(db.Model):
    # ...
    password_hash = db.Column(db.String(128))

    @property
    def password(self):
        raise AttributeError('password is not a readable attribute')

    @password.setter
    def password(self, password):
        self.password_hash = generate_password_hash(password)

    def verify_password(self, password):
        return check_password_hash(self.password_hash, password)
```

The password hashing function is implemented through a write-only property called `password`. When this property is set, the setter method will call Werkzeug's `generate_password_hash()` function and write the result to the `password_hash` field. Attempting to read the `password` property will return an error, as clearly the original password cannot be recovered once hashed.

The `verify_password()` method takes a password and passes it to Werkzeug's `check_password_hash()` function for verification against the hashed version stored in the `User` model. If this method returns `True`, then the password is correct.

> If you have cloned the application's Git repository on GitHub, you can run `git checkout 8a` to check out this version of the application.

The password hashing functionality is now complete and can be tested in the shell:

```
(venv) $ flask shell
>>> u = User()
>>> u.password = 'cat'
>>> u.password
Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File "/home/flask/flasky/app/models.py", line 24, in password
    raise AttributeError('password is not a readable attribute')
AttributeError: password is not a readable attribute
>>> u.password_hash
'pbkdf2:sha256:50000$moHwFH1B$ef1574909f9c549285e8547cad181c5e0213cfa44a4aba4349
fa830aa1fd227f'
>>> u.verify_password('cat')
True
>>> u.verify_password('dog')
False
>>> u2 = User()
>>> u2.password = 'cat'
>>> u2.password_hash
'pbkdf2:sha256:50000$Pfz0m0KU$27be930b7f0e0119d38e8d8a62f7f5e75c0a7db61ae16709bc
aa6cfd60c44b74'
```

Note how trying to access the `password` property of a user returns an `AttributeError`. Also, users `u` and `u2` have completely different password hashes, even though they both use the same password. To ensure that this functionality continues to work in the future, the preceding tests done manually can be written as unit tests that can be repeated easily. In Example 8-2 a new module inside the *tests* package is shown with three new tests that exercise the recent changes to the `User` model.

Example 8-2. tests/test_user_model.py: password hashing tests

```
import unittest
from app.models import User

class UserModelTestCase(unittest.TestCase):
    def test_password_setter(self):
        u = User(password = 'cat')
        self.assertTrue(u.password_hash is not None)

    def test_no_password_getter(self):
        u = User(password = 'cat')
        with self.assertRaises(AttributeError):
            u.password

    def test_password_verification(self):
        u = User(password = 'cat')
        self.assertTrue(u.verify_password('cat'))
        self.assertFalse(u.verify_password('dog'))
```

```python
    def test_password_salts_are_random(self):
        u = User(password='cat')
        u2 = User(password='cat')
        self.assertTrue(u.password_hash != u2.password_hash)
```

To run these new unit tests, use the following command:

```
(venv) $ flask test
test_app_exists (test_basics.BasicsTestCase) ... ok
test_app_is_testing (test_basics.BasicsTestCase) ... ok
test_no_password_getter (test_user_model.UserModelTestCase) ... ok
test_password_salts_are_random (test_user_model.UserModelTestCase) ... ok
test_password_setter (test_user_model.UserModelTestCase) ... ok
test_password_verification (test_user_model.UserModelTestCase) ... ok

.----------------------------------------------------------------------
Ran 6 tests in 0.379s

OK
```

You can run the unit test suite like this every time you want to confirm everything is working as expected. Having the automation in place makes verifying this feature very low cost, so testing should be repeated often, to ensure that this functionality does not break in the future.

Creating an Authentication Blueprint

Blueprints were introduced in Chapter 7 as a way to define routes in the global scope after the creation of the application was moved into a factory function. In this section, the routes related to the user authentication subsystem will be added to a second blueprint, called `auth`. Using different blueprints for different subsystems of the application is a great way to keep the code neatly organized.

The `auth` blueprint will be hosted in a Python package with the same name. The blueprint's package constructor creates the blueprint object and imports routes from a *views.py* module. This is shown in Example 8-3.

Example 8-3. app/auth/__init__.py: authentication blueprint creation

```python
from flask import Blueprint

auth = Blueprint('auth', __name__)

from . import views
```

The *app/auth/views.py* module, shown in Example 8-4, imports the blueprint and defines the routes associated with authentication using its `route` decorator. For now, a `/login` route is added, which renders a placeholder template of the same name.

Example 8-4. app/auth/views.py: authentication blueprint routes and view functions

```
from flask import render_template
from . import auth

@auth.route('/login')
def login():
    return render_template('auth/login.html')
```

Note that the template file given to `render_template()` is stored inside the *auth* directory. This directory must be created inside *app/templates*, as Flask expects the templates' paths to be relative to the application's templates directory. By storing the blueprint templates in their own subdirectory, there is no risk of naming collisions with the `main` blueprint or any other blueprints that will be added in the future.

> Blueprints can also be configured to have their own independent directories for templates. When multiple template directories have been configured, the `render_template()` function searches the templates directory configured for the application first, and then searches the template directories defined by blueprints.

The `auth` blueprint needs to be attached to the application in the `create_app()` factory function, as shown in Example 8-5.

Example 8-5. app/__init__.py: authentication blueprint registration

```
def create_app(config_name):
    # ...
    from .auth import auth as auth_blueprint
    app.register_blueprint(auth_blueprint, url_prefix='/auth')

    return app
```

The `url_prefix` argument in the blueprint registration is optional. When used, all the routes defined in the blueprint will be registered with the given prefix, in this case */auth*. For example, the */login* route will be registered as */auth/login*, and the fully qualified URL under the development web server then becomes *http://localhost:5000/auth/login*.

> If you have cloned the application's Git repository on GitHub, you can run `git checkout 8b` to check out this version of the application.

User Authentication with Flask-Login

When users log in to the application, their authenticated state has to be recorded in the user session, so that it is remembered as they navigate through different pages. Flask-Login is a small but extremely useful extension that specializes in managing this particular aspect of a user authentication system, without being tied to a specific authentication mechanism.

To begin, the extension needs to be installed in the virtual environment:

```
(venv) $ pip install flask-login
```

Preparing the User Model for Logins

Flask-Login works closely with the application's own `User` objects. To be able to work with the application's `User` model, the Flask-Login extension requires it to implement a few common properties and methods. The required items are shown in Table 8-1.

Table 8-1. Flask-Login required items

Property/method	Description
`is_authenticated`	Must be `True` if the user has valid login credentials or `False` otherwise.
`is_active`	Must be `True` if the user is allowed to log in or `False` otherwise. A `False` value can be used for disabled accounts.
`is_anonymous`	Must always be `False` for regular users and `True` for a special user object that represents anonymous users.
`get_id()`	Must return a unique identifier for the user, encoded as a Unicode string.

These properties and methods can be implemented directly in the model class, but as an easier alternative Flask-Login provides a `UserMixin` class that has default implementations that are appropriate for most cases. The updated `User` model is shown in Example 8-6.

Example 8-6. app/models.py: updates to the User model to support user logins

```python
from flask_login import UserMixin

class User(UserMixin, db.Model):
    __tablename__ = 'users'
    id = db.Column(db.Integer, primary_key = True)
    email = db.Column(db.String(64), unique=True, index=True)
    username = db.Column(db.String(64), unique=True, index=True)
    password_hash = db.Column(db.String(128))
    role_id = db.Column(db.Integer, db.ForeignKey('roles.id'))
```

Note that an `email` field was also added. In this application, users will log in with their email addresses, as they are less likely to forget those than their usernames.

Flask-Login is initialized in the application factory function, as shown in Example 8-7.

Example 8-7. app/__init__.py: Flask-Login initialization

```
from flask_login import LoginManager

login_manager = LoginManager()
login_manager.login_view = 'auth.login'

def create_app(config_name):
    # ...
    login_manager.init_app(app)
    # ...
```

The `login_view` attribute of the `LoginManager` object sets the endpoint for the login page. Flask-Login will redirect to the login page when an anonymous user tries to access a protected page. Because the login route is inside a blueprint, it needs to be prefixed with the blueprint name.

Finally, Flask-Login requires the application to designate a function to be invoked when the extension needs to load a user from the database given its identifier. This function is shown in Example 8-8.

Example 8-8. app/models.py: user loader function

```
from . import login_manager

@login_manager.user_loader
def load_user(user_id):
    return User.query.get(int(user_id))
```

The `login_manager.user_loader` decorator is used to register the function with Flask-Login, which will call it when it needs to retrieve information about the logged-in user. The user identifier will be passed as a string, so the function converts it to an integer before it passes it to the Flask-SQLAlchemy query that loads the user. The return value of the function must be the user object, or `None` if the user identifier is invalid or any other error occurred.

Protecting Routes

To protect a route so that it can only be accessed by authenticated users, Flask-Login provides a `login_required` decorator. An example of its usage follows:

```python
from flask_login import login_required

@app.route('/secret')
@login_required
def secret():
    return 'Only authenticated users are allowed!'
```

You can see from this example that it is possible to "chain" multiple function decorators. When two or more decorators are added to a function, each decorator only affects those that are below it, in addition to the target function. In this example, the `secret()` function will be protected against unauthorized users with `login_required`, and then the resulting function will be registered with Flask as a route. Reversing the order will produce the wrong result, as the original function will be registered as a route before it receives the additional properties from the `login_required` decorator.

Thanks to the `login_required` decorator, if this route is accessed by a user who is not authenticated, Flask-Login will intercept the request and send the user to the login page instead.

Adding a Login Form

The login form that will be presented to users has a text field for the email address, a password field, a "remember me" checkbox, and a submit button. The Flask-WTF form class that defines this form is shown in Example 8-9.

Example 8-9. app/auth/forms.py: login form

```python
from flask_wtf import FlaskForm
from wtforms import StringField, PasswordField, BooleanField, SubmitField
from wtforms.validators import DataRequired, Length, Email

class LoginForm(FlaskForm):
    email = StringField('Email', validators=[DataRequired(), Length(1, 64),
                                             Email()])
    password = PasswordField('Password', validators=[DataRequired()])
    remember_me = BooleanField('Keep me logged in')
    submit = SubmitField('Log In')
```

The `PasswordField` class represents an `<input>` element with `type="password"`. The `BooleanField` class represents a checkbox.

The email field uses the `Length()` and `Email()` validators from WTForms in addition to `DataRequired()`, to ensure that the user not only provides a value for this field, but that it is valid. When providing a list of validators, WTForms will evaluate them in the order provided, and in case of a validation failure the error message shown will be the one of the first validator that failed.

The template associated with the login page is stored in *auth/login.html*. This template just needs to render the form using Flask-Bootstrap's `wtf.quick_form()` macro. Figure 8-1 shows the login form rendered by the web browser.

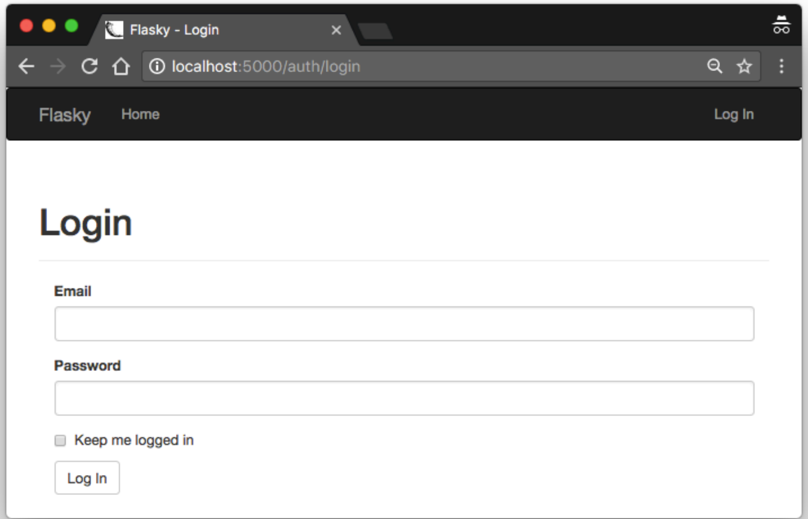


Figure 8-1. Login form

The navigation bar in the *base.html* template uses a Jinja2 conditional to display "Log In" or "Log Out" links depending on the logged-in state of the current user. The conditional is shown in Example 8-10.

Example 8-10. app/templates/base.html: Log In and Log Out navigation bar links

```
<ul class="nav navbar-nav navbar-right">
    {% if current_user.is_authenticated %}
    <li><a href="{{ url_for('auth.logout') }}">Log Out</a></li>
    {% else %}
    <li><a href="{{ url_for('auth.login') }}">Log In</a></li>
    {% endif %}
</ul>
```

The `current_user` variable used in the conditional is defined by Flask-Login and is automatically available to view functions and templates. This variable contains the currently logged-in user, or a proxy anonymous user object if the user is not logged in. Anonymous user objects have the `is_authenticated` property set to `False`, so the

expression current_user.is_authenticated is a convenient way to know whether the current user is logged in.

Signing Users In

The implementation of the login() view function is shown in Example 8-11.

Example 8-11. app/auth/views.py: login route

```
from flask import render_template, redirect, request, url_for, flash
from flask_login import login_user
from . import auth
from ..models import User
from .forms import LoginForm

@auth.route('/login', methods=['GET', 'POST'])
def login():
    form = LoginForm()
    if form.validate_on_submit():
        user = User.query.filter_by(email=form.email.data).first()
        if user is not None and user.verify_password(form.password.data):
            login_user(user, form.remember_me.data)
            next = request.args.get('next')
            if next is None or not next.startswith('/'):
                next = url_for('main.index')
            return redirect(next)
        flash('Invalid username or password.')
    return render_template('auth/login.html', form=form)
```

The view function creates a LoginForm object and uses it like the simple form in Chapter 4. When the request is of type GET, the view function just renders the template, which in turn displays the form. When the form is submitted in a POST request, Flask-WTF's validate_on_submit() function validates the form variables, and then attempts to log the user in.

To log a user in, the function begins by loading the user from the database using the email provided with the form. If a user with the given email address exists, then its verify_password() method is called with the password that also came with the form. If the password is valid, Flask-Login's login_user() function is invoked to record the user as logged in for the user session. The login_user() function takes the user to log in and an optional "remember me" Boolean, which was also submitted with the form. A value of False for this argument causes the user session to expire when the browser window is closed, so the user will have to log in again next time. A value of True causes a long-term cookie to be set in the user's browser, which Flask-Login uses to restore the user session. The optional REMEMBER_COOKIE_DURATION configuration option can be used to change the default one-year duration for the remember cookie.

In accordance with the Post/Redirect/Get pattern discussed in Chapter 4, the `POST` request that submitted the login credentials ends with a redirect, but there are two possible URL destinations. If the login form was presented to the user to prevent unauthorized access to a protected URL the user wanted to visit, then Flask-Login will have saved that original URL in the `next` query string argument, which can be accessed from the `request.args` dictionary. If the `next` query string argument is not available, a redirect to the home page is issued instead. The URL in `next` is validated to make sure it is a relative URL, to prevent a malicious user from using this argument to redirect unsuspecting users to another site.

For the case where the email address or password provided by the user is invalid, a flash message is set and the form is rendered again for the user to retry.

> On a production server, the application must be made available over secure HTTP, so that login credentials and user sessions are always transmitted encrypted. Without secure HTTP, sensitive data can be intercepted during transit by an attacker.

The login template needs to be updated to render the form. These changes are shown in Example 8-12.

Example 8-12. app/templates/auth/login.html: login form template

```
{% extends "base.html" %}
{% import "bootstrap/wtf.html" as wtf %}

{% block title %}Flasky - Login{% endblock %}

{% block page_content %}
<div class="page-header">
    <h1>Login</h1>
</div>
<div class="col-md-4">
    {{ wtf.quick_form(form) }}
</div>
{% endblock %}
```

Signing Users Out

The implementation of the logout route is shown in Example 8-13.

Example 8-13. app/auth/views.py: logout route

```
from flask_login import logout_user, login_required

@auth.route('/logout')
```

```
@login_required
def logout():
    logout_user()
    flash('You have been logged out.')
    return redirect(url_for('main.index'))
```

To log a user out, Flask-Login's `logout_user()` function is called to remove and reset the user session. The logout is completed with a flash message that confirms the action and a redirect to the home page.

> If you have cloned the application's Git repository on GitHub, you can run `git checkout 8c` to check out this version of the application. This update contains a database migration, so remember to run `flask db upgrade` after you check out the code. To ensure that you have all the dependencies installed, also run `pip install -r requirements.txt`.

Understanding How Flask-Login Works

Flask-Login is a fairly small extension, but due to the many moving pieces involved in the authentication flow, Flask users often have trouble understanding how the extension works. The following is the sequence of operations that occur when a user logs in to the system:

1. The user navigates to *http://localhost:5000/auth/login* by clicking on the "Log In" link. The handler for this URL returns the login form template.
2. The user enters their username and password, and presses the Submit button. The same handler is invoked again, but now as a `POST` request instead of `GET`.
    a. The handler validates the credentials submitted with the form, and then invokes Flask-Login's `login_user()` function to log the user in.
    b. The `login_user()` function writes the ID of the user to the user session as a string.
    c. The view function returns with a redirect to the home page.
3. The browser receives the redirect and requests the home page.
    a. The view function for the home page is invoked, and it triggers the rendering of the main Jinja2 template.
    b. During the rendering of the Jinja2 template, a reference to Flask-Login's `current_user` appears for the first time.
    c. The `current_user` context variable does not have a value assigned for this request yet, so it invokes Flask-Login's internal function `_get_user()` to find out who the user is.

d. The `_get_user()` function checks if there is a user ID stored in the user session. If there isn't one, it returns an instance of Flask-Login's `AnonymousUser`. If there is an ID, it invokes the function that the application registered with the `user_loader` decorator, with the ID as its argument.
e. The application's `user_loader` handler reads the user from the database and returns it. Flask-Login assigns it to the `current_user` context variable for the current request.
f. The template receives the newly assigned value of `current_user`.

The `login_required` decorator builds on top of the `current_user` context variable by only allowing the decorated view function to run when the expression `current_user.is_authenticated` is `True`. The `logout_user()` function simply deletes the user ID from the user session.

Testing Logins

To verify that the login functionality is working, the home page can be updated to greet the logged-in user by name. The template section that generates the greeting is shown in Example 8-14.

Example 8-14. app/templates/index.html: greeting the logged-in user

```
Hello,
{% if current_user.is_authenticated %}
    {{ current_user.username }}
{% else %}
    Stranger
{% endif %}!
```

In this template once again `current_user.is_authenticated` is used to determine whether the user is logged in.

Because no user registration functionality has been built, a new user can only be registered from the shell at this time:

```
(venv) $ $ flask shell
>>> u = User(email='john@example.com', username='john', password='cat')
>>> db.session.add(u)
>>> db.session.commit()
```

The user created previously can now log in. Figure 8-2 shows the application home page with the user logged in.

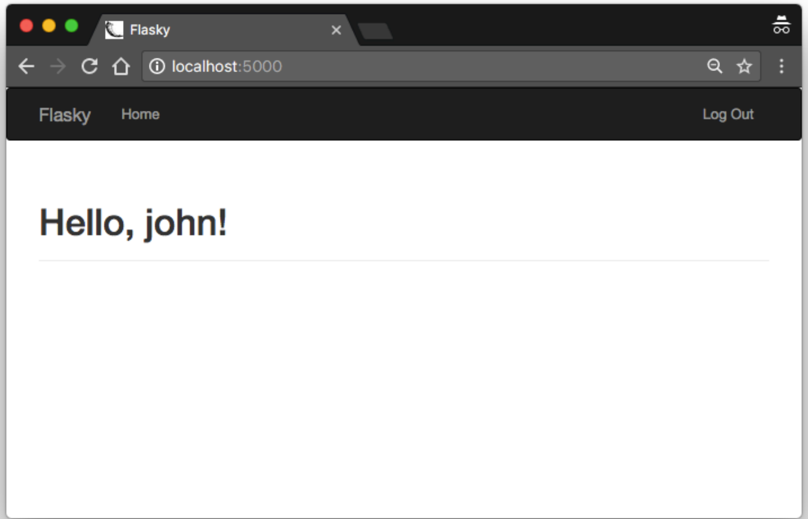


Figure 8-2. Home page after successful login

New User Registration

When new users want to become members of the application, they must register with it so that they are known and can log in. A link in the login page will send them to a registration page, where they can enter their email address, username, and password.

Adding a User Registration Form

The form that will be used in the registration page asks the user to enter an email address, username, and password. This form is shown in Example 8-15.

Example 8-15. app/auth/forms.py: user registration form

```
from flask_wtf import FlaskForm
from wtforms import StringField, PasswordField, BooleanField, SubmitField
from wtforms.validators import DataRequired, Length, Email, Regexp, EqualTo
from wtforms import ValidationError
from ..models import User

class RegistrationForm(FlaskForm):
    email = StringField('Email', validators=[DataRequired(), Length(1, 64),
                                             Email()])
```

```
    username = StringField('Username', validators=[
        DataRequired(), Length(1, 64),
        Regexp('^[A-Za-z][A-Za-z0-9_.]*$', 0,
               'Usernames must have only letters, numbers, dots or '
               'underscores')])
    password = PasswordField('Password', validators=[
        DataRequired(), EqualTo('password2', message='Passwords must match.')])
    password2 = PasswordField('Confirm password', validators=[DataRequired()])
    submit = SubmitField('Register')

    def validate_email(self, field):
        if User.query.filter_by(email=field.data).first():
            raise ValidationError('Email already registered.')

    def validate_username(self, field):
        if User.query.filter_by(username=field.data).first():
            raise ValidationError('Username already in use.')
```

This form uses the `Regexp` validator from WTForms to ensure that the `username` field starts with a letter and only contains letters, numbers, underscores, and dots. The two arguments to the validator that follow the regular expression are the regular expression flags and the error message to display on failure.

The password is entered twice as a safety measure, but this step makes it necessary to validate that the two password fields have the same content, which is done with another validator from WTForms called `EqualTo`. This validator is attached to one of the password fields with the name of the other field given as an argument.

This form also has two custom validators implemented as methods. When a form defines a method with the prefix `validate_` followed by the name of a field, the method is invoked in addition to any regularly defined validators. In this case, the custom validators for `email` and `username` ensure that the values given are not duplicates. The custom validators indicate a validation error by raising a `ValidationError` exception with the text of the error message as an argument.

The template that presents this form is called */templates/auth/register.html*. Like the login template, this one also renders the form with `wtf.quick_form()`. The registration page is shown in Figure 8-3.

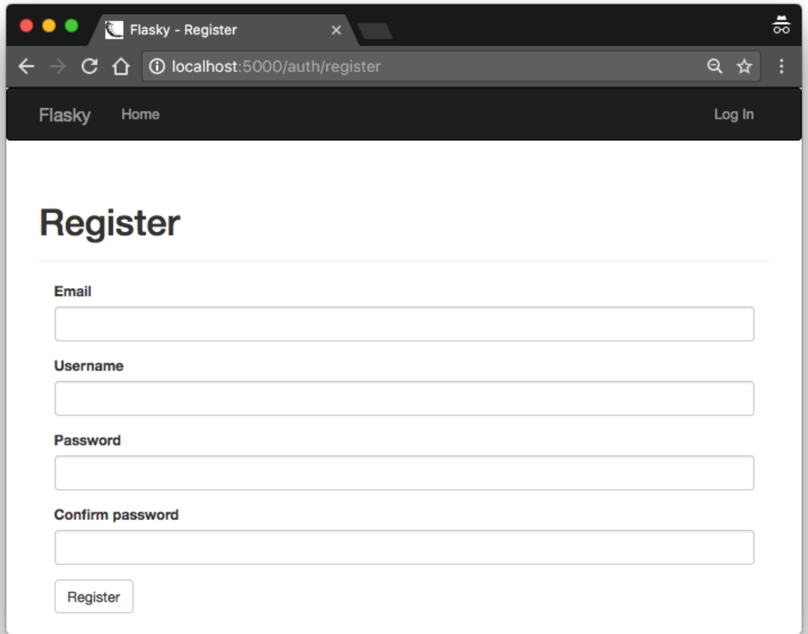


Figure 8-3. New user registration form

The registration page needs to be linked from the login page so that users who don't have an account can easily find it. This change is shown in Example 8-16.

Example 8-16. app/templates/auth/login.html: link to the registration page

```
<p>
    New user?
    <a href="{{ url_for('auth.register') }}">
        Click here to register
    </a>
</p>
```

Registering New Users

Handling user registrations does not present any big surprises. When the registration form is submitted and validated, a new user is added to the database using the information provided by the user. The view function that performs this task is shown in Example 8-17.

Example 8-17. app/auth/views.py: user registration route

```
@auth.route('/register', methods=['GET', 'POST'])
def register():
    form = RegistrationForm()
    if form.validate_on_submit():
        user = User(email=form.email.data,
                    username=form.username.data,
                    password=form.password.data)
        db.session.add(user)
        db.session.commit()
        flash('You can now login.')
        return redirect(url_for('auth.login'))
    return render_template('auth/register.html', form=form)
```

If you have cloned the application's Git repository on GitHub, you can run `git checkout 8d` to check out this version of the application.

Account Confirmation

For certain types of applications, it is important to ensure that the user information provided during registration is valid. A common requirement is to ensure that the user can be reached through the provided email address.

To validate the email address, applications send a confirmation email to users immediately after they register. The new account is initially marked as unconfirmed until the instructions in the email are followed, which proves that the user has received the email. The account confirmation procedure usually involves clicking a specially crafted URL link that includes a confirmation token.

Generating Confirmation Tokens with itsdangerous

The simplest account confirmation link would be a URL with the format *http://www.example.com/auth/confirm/<id>* included in the confirmation email, where *<id>* is the numeric `id` assigned to the user in the database. When the user clicks the link, the view function that handles this route receives the user `id` to confirm as an argument and can easily update the confirmed status of the user.

But this is obviously not a secure implementation, as any user who figures out the format of the confirmation links will be able to confirm arbitrary accounts just by sending random numbers in the URL. The idea is to replace the *<id>* in the URL with a token that contains the same information, but in such a way that only the server can generate valid confirmation URLs.

If you recall the discussion on user sessions in Chapter 4, Flask uses cryptographically signed cookies to protect the content of user sessions against tampering. The user session cookies contain a cryptographic signature generated by a package called `itsdangerous`. If the contents of the user session is altered, the signature will not match the content anymore, so Flask discards the session and starts a new one. The same concept can be applied to confirmation tokens.

The following is a short shell session that shows how `itsdangerous` can generate a signed token that contains a user `id` inside:

```
(venv) $ flask shell
>>> from itsdangerous import TimedJSONWebSignatureSerializer as Serializer
>>> s = Serializer(app.config['SECRET_KEY'], expires_in=3600)
>>> token = s.dumps({ 'confirm': 23 })
>>> token
'eyJhbGciOiJIUzI1NiIsImV4cCI6MTM4MTcxODU1OCwiaWF0IjoxMzgxNzE0OTU4fQ.ey ...'
>>> data = s.loads(token)
>>> data
{'confirm': 23}
```

The `itsdangerous` package provides several types of token generators. Among them, the class `TimedJSONWebSignatureSerializer` generates JSON Web Signatures (JWSs) with a time expiration. The constructor of this class takes an encryption key as an argument, which in a Flask application can be the configured `SECRET_KEY`.

The `dumps()` method generates a cryptographic signature for the data given as an argument and then serializes the data plus the signature as a convenient token string. The `expires_in` argument sets an expiration time for the token, expressed in seconds.

To decode the token, the serializer object provides a `loads()` method that takes the token as its only argument. The function verifies the signature and the expiration time and, if both are valid, it returns the original data. When the `loads()` method is given an invalid token or a valid token that is expired, an exception is raised.

Token generation and verification using this functionality can be added to the `User` model. The changes are shown in Example 8-18.

Example 8-18. app/models.py: user account confirmation

```python
from itsdangerous import TimedJSONWebSignatureSerializer as Serializer
from flask import current_app
from . import db

class User(UserMixin, db.Model):
    # ...
    confirmed = db.Column(db.Boolean, default=False)
```

```
    def generate_confirmation_token(self, expiration=3600):
        s = Serializer(current_app.config['SECRET_KEY'], expiration)
        return s.dumps({'confirm': self.id}).decode('utf-8')

    def confirm(self, token):
        s = Serializer(current_app.config['SECRET_KEY'])
        try:
            data = s.loads(token.encode('utf-8'))
        except:
            return False
        if data.get('confirm') != self.id:
            return False
        self.confirmed = True
        db.session.add(self)
        return True
```

The `generate_confirmation_token()` method generates a token with a default validity time of one hour. The `confirm()` method verifies the token and, if valid, sets the new `confirmed` attribute in the user model to `True`.

In addition to verifying the token, the `confirm()` function checks that the `id` from the token matches the logged-in user, which is stored in `current_user`. This ensures that a confirmation token for a given user cannot be used to confirm a different user.

> Because a new column was added to the model to track the confirmed state of each account, a new database migration needs to be generated and applied.

The two new methods added to the `User` model are easily tested in unit tests. You can find the unit tests in the GitHub repository for the application.

Sending Confirmation Emails

The current */register* route redirects to */index* after adding the new user to the database. Before redirecting, this route now needs to send the confirmation email. This change is shown in Example 8-19.

Example 8-19. app/auth/views.py: registration route with confirmation email

```
from ..email import send_email

@auth.route('/register', methods=['GET', 'POST'])
def register():
    form = RegistrationForm()
    if form.validate_on_submit():
        # ...
```

```
        db.session.add(user)
        db.session.commit()
        token = user.generate_confirmation_token()
        send_email(user.email, 'Confirm Your Account',
                   'auth/email/confirm', user=user, token=token)
        flash('A confirmation email has been sent to you by email.')
        return redirect(url_for('main.index'))
    return render_template('auth/register.html', form=form)
```

Note that a `db.session.commit()` call had to be added before the confirmation email is sent out. The problem is that new users get assigned an `id` when they are committed to the database, and this `id` is needed to generate the confirmation token.

The email templates used by the authentication blueprint will be added in the *templates/auth/email* directory to keep them separate from the HTML templates. As discussed in Chapter 6, for each email two templates are needed for the plain-text and HTML versions of the body. As an example, Example 8-20 shows the plain-text version of the confirmation email template, and you can find the equivalent HTML version in the GitHub repository.

Example 8-20. app/templates/auth/email/confirm.txt: text body of confirmation email

```
Dear {{ user.username }},

Welcome to Flasky!

To confirm your account please click on the following link:

{{ url_for('auth.confirm', token=token, _external=True) }}

Sincerely,

The Flasky Team

Note: replies to this email address are not monitored.
```

By default, `url_for()` generates relative URLs; so, for example, `url_for('auth.confirm', token='abc')` returns the string `'/auth/confirm/abc'`. This, of course, is not a valid URL that can be sent in an email, since it is only the path portion of the URL. Relative URLs work fine when they are used within the context of a web page because the browser converts them to absolute URLs by adding the hostname and port number from the current page, but when sending a URL over email there is no such context. The `_external=True` argument is added to the `url_for()` call to request a fully qualified URL that includes the scheme (*http://* or *https://*), hostname, and port.

The view function that confirms accounts is shown in Example 8-21.

Example 8-21. app/auth/views.py: confirming a user account

```
from flask_login import current_user

@auth.route('/confirm/<token>')
@login_required
def confirm(token):
    if current_user.confirmed:
        return redirect(url_for('main.index'))
    if current_user.confirm(token):
        db.session.commit()
        flash('You have confirmed your account. Thanks!')
    else:
        flash('The confirmation link is invalid or has expired.')
    return redirect(url_for('main.index'))
```

This route is protected with the `login_required` decorator from Flask-Login, so that when the users click on the link from the confirmation email they are asked to log in before they reach this view function.

The function first checks if the logged-in user is already confirmed, and in that case it redirects to the home page, as obviously there is nothing to do. This can prevent unnecessary work if a user clicks the confirmation token multiple times by mistake.

Because the actual token confirmation is done entirely in the `User` model, all the view function needs to do is call the `confirm()` method and then flash a message according to the result. When the confirmation succeeds, the `User` model's `confirmed` attribute is changed and added to the session and then the database session is committed.

Each application can decide what unconfirmed users are allowed to do before they confirm their accounts. One possibility is to allow unconfirmed users to log in, but only show them a page that asks them to confirm their accounts before they can gain further access.

This step can be done using Flask's `before_request` hook, which was briefly described in Chapter 2. From a blueprint, the `before_request` hook applies only to requests that belong to the blueprint. To install a blueprint hook for all application requests, the `before_app_request` decorator must be used instead. Example 8-22 shows how this handler is implemented.

Example 8-22. app/auth/views.py: filtering unconfirmed accounts with the before_app_request handler

```
@auth.before_app_request
def before_request():
    if current_user.is_authenticated \
            and not current_user.confirmed \
            and request.blueprint != 'auth' \
            and request.endpoint != 'static':
        return redirect(url_for('auth.unconfirmed'))

@auth.route('/unconfirmed')
def unconfirmed():
    if current_user.is_anonymous or current_user.confirmed:
        return redirect(url_for('main.index'))
    return render_template('auth/unconfirmed.html')
```

The `before_app_request` handler will intercept a request when three conditions are true:

1. A user is logged in (`current_user.is_authenticated` is `True`).
2. The account for the user is not confirmed.
3. The requested URL is outside of the authentication blueprint and is not for a static file. Access to the authentication routes needs to be granted, as those are the routes that will enable the user to confirm the account or perform other account management functions.

If these three conditions are met, then a redirect is issued to a new */auth/unconfirmed* route that shows a page with information about account confirmation.

> When a `before_request` or `before_app_request` callback returns a response or a redirect, Flask sends that to the client without invoking the view function associated with the request. This effectively allows these callbacks to intercept a request when necessary.

The page that is presented to unconfirmed users (shown in Figure 8-4) just renders a template that gives users instructions for how to confirm their accounts and offers a link to request a new confirmation email, in case the original email was lost. The route that resends the confirmation email is shown in Example 8-23.

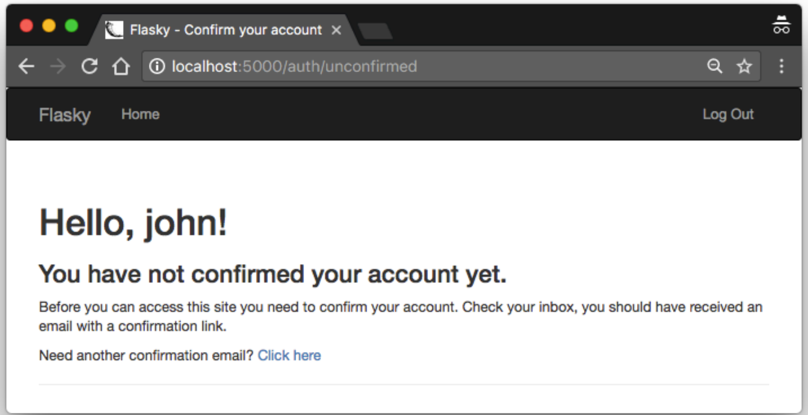


Figure 8-4. Unconfirmed account page

Example 8-23. app/auth/views.py: resending the account confirmation email

```
@auth.route('/confirm')
@login_required
def resend_confirmation():
    token = current_user.generate_confirmation_token()
    send_email(current_user.email, 'Confirm Your Account',
               'auth/email/confirm', user=current_user, token=token)
    flash('A new confirmation email has been sent to you by email.')
    return redirect(url_for('main.index'))
```

This route repeats what was done in the registration route using `current_user`, the user who is logged in, as the target user. This route is also protected with `login_required` to ensure that when it is accessed, the user that is making the request is authenticated.

> If you have cloned the application's Git repository on GitHub, you can run `git checkout 8e` to check out this version of the application. This update contains a database migration, so remember to run `flask db upgrade` after you check out the code.

Account Management

Users who have accounts with the application may need to make changes to their accounts from time to time. The following tasks can be added to the authentication blueprint using the techniques presented in this chapter:

Password updates
: Security-conscious users may want to change their passwords periodically. This is an easy feature to implement, because as long as the user is logged in, it is safe to present a form that asks for the old password and a new password to replace it. This feature is implemented as commit `8f` in the GitHub repository. As part of this change, the "Log Out" link in the navigation bar was refactored into a dropdown that contains the "Change Password" and "Log Out" links.

Password resets
: To avoid locking users out of the application when they forget their passwords, a password reset option can be offered. To implement password resets in a secure way, it is necessary to use tokens similar to those used to confirm accounts. When a user requests a password reset, an email with a reset token is sent to the registered email address. The user then clicks the link in the email and, after the token is verified, a form is presented where a new password can be entered. This feature is implemented as commit `8g` in the GitHub repository.

Email address changes
: Users can be given the option to change their registered email address, but before the new address is accepted it must be verified with a confirmation email. To use this feature, the user enters the new email address in a form. To confirm the email address, a token is emailed to that address. When the server receives the token back, it can update the user object. While the server waits to receive the token, it can store the new email address in a new database field reserved for pending email addresses, or it can store the address in the token along with the `id`. This feature is implemented as commit `8h` in the GitHub repository.

In the next chapter, the user subsystem of Flasky will be extended through the use of user roles.

CHAPTER 9
User Roles

Not all users of web applications are created equal. In most applications, a small percentage of users are trusted with extra powers to help keep the application running smoothly. Administrators are the best example, but in many cases middle-level power users such as content moderators exist as well. To implement this, all users are assigned a *role*.

There are several ways to implement roles in an application. The appropriate method largely depends on how many roles need to be supported and how elaborate they are. For example, a simple application may need just two roles, one for regular users and one for administrators. In this case, having an `is_administrator` Boolean field in the `User` model may be all that is necessary. A more complex application may need additional roles with varying levels of power in between regular users and administrators. In some applications it may not even make sense to talk about discrete roles, and instead giving users a set of individual *permissions* may be the right approach.

The user role implementation presented in this chapter is a hybrid between discrete roles and permissions. Users are assigned a discrete role, but each role defines what actions it allows its users to perform through a list of permissions.

Database Representation of Roles

A simple `roles` table was created in Chapter 5 as a vehicle to demonstrate one-to-many relationships. Example 9-1 shows an improved `Role` model with some additions.

Example 9-1. app/models.py: role database model

```
class Role(db.Model):
    __tablename__ = 'roles'
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(64), unique=True)
    default = db.Column(db.Boolean, default=False, index=True)
    permissions = db.Column(db.Integer)
    users = db.relationship('User', backref='role', lazy='dynamic')

    def __init__(self, **kwargs):
        super(Role, self).__init__(**kwargs)
        if self.permissions is None:
            self.permissions = 0
```

The `default` field is one of the additions to this model. This field should be set to `True` for only one role and `False` for all the others. The role marked as default will be the one assigned to new users upon registration. Since the application is going to search the `roles` table to find the default one, this column is configured to have an index, as that will make searches much faster.

Another addition to the model is the `permissions` field, which is an integer value that defines the list of permissions for the role in a compact way. Since SQLAlchemy will set this field to `None` by default, a class constructor is added that sets it to `0` if an initial value isn't provided in the constructor arguments.

The list of tasks for which permissions are needed is obviously application specific. For Flasky, the list is shown in Table 9-1.

Table 9-1. Application permissions

Task name	Permission name	Permission value
Follow users	`FOLLOW`	1
Comment on posts made by others	`COMMENT`	2
Write articles	`WRITE`	4
Moderate comments made by others	`MODERATE`	8
Administration access	`ADMIN`	16

The benefit of using powers of two for permission values is that it allows permissions to be combined, giving each possible combination of permissions a unique value to store in the role's `permissions` field. For example, for a user role that gives users permission to follow other users and comment on posts, the permission value is `FOLLOW + COMMENT = 3`. This is a very efficient way to store the list of permissions assigned to each role.

The code representation of Table 9-1 is shown in Example 9-2.

Example 9-2. app/models.py: permission constants

```
class Permission:
    FOLLOW = 1
    COMMENT = 2
    WRITE = 4
    MODERATE = 8
    ADMIN = 16
```

With the permission constants in place, a few new methods can be added to the `Role` model to manage permissions. These are shown in Example 9-3.

Example 9-3. app/models.py: permission management in the Role model

```
class Role(db.Model):
    # ...

    def add_permission(self, perm):
        if not self.has_permission(perm):
            self.permissions += perm

    def remove_permission(self, perm):
        if self.has_permission(perm):
            self.permissions -= perm

    def reset_permissions(self):
        self.permissions = 0

    def has_permission(self, perm):
        return self.permissions & perm == perm
```

The `add_permission()`, `remove_permission()`, and `reset_permission()` methods all use basic arithmetic operations to update the permission list. The `has_permission()` method is the most complex of the set, as it relies on the bitwise and operator `&` to check if a combined permission value includes the given basic permission. You can play with these methods in a Python shell:

```
(venv) $ flask shell
>>> r = Role(name='User')
>>> r.add_permission(Permission.FOLLOW)
>>> r.add_permission(Permission.WRITE)
>>> r.has_permission(Permission.FOLLOW)
True
>>> r.has_permission(Permission.ADMIN)
False
>>> r.reset_permissions()
```

```
>>> r.has_permission(Permission.FOLLOW)
False
```

Table 9-2 shows the list of user roles that will be supported in this application, along with the permission combinations that define each of them.

Table 9-2. User roles

User role	Permissions	Description
None	None	Read-only access to the application. This applies to unknown users who are not logged in.
User	`FOLLOW`, `COMMENT`, `WRITE`	Basic permissions to write articles and comments and to follow other users. This is the default for new users.
Moderator	`FOLLOW`, `COMMENT`, `WRITE`, `MODERATE`	Adds permission to moderate comments made by other users.
Administrator	`FOLLOW`, `COMMENT`, `WRITE`, `MODERATE`, `ADMIN`	Full access, which includes permission to change the roles of other users.

Adding the roles to the database manually is time consuming and error prone, so instead a class method can be added to the `Role` class for this purpose, as shown in Example 9-4. This will make it easy to re-create the correct roles and permissions during unit testing and, more importantly, on the production server once the application is deployed.

Example 9-4. app/models.py: creating roles in the database

```
class Role(db.Model):
    # ...
    @staticmethod
    def insert_roles():
        roles = {
            'User': [Permission.FOLLOW, Permission.COMMENT, Permission.WRITE],
            'Moderator': [Permission.FOLLOW, Permission.COMMENT,
                          Permission.WRITE, Permission.MODERATE],
            'Administrator': [Permission.FOLLOW, Permission.COMMENT,
                              Permission.WRITE, Permission.MODERATE,
                              Permission.ADMIN],
        }
        default_role = 'User'
        for r in roles:
            role = Role.query.filter_by(name=r).first()
            if role is None:
                role = Role(name=r)
            role.reset_permissions()
            for perm in roles[r]:
                role.add_permission(perm)
            role.default = (role.name == default_role)
```

```
            db.session.add(role)
        db.session.commit()
```

The `insert_roles()` function does not directly create new role objects. Instead, it tries to find existing roles by name and update those. A new role object is created only for roles that aren't in the database already. This is done so that the role list can be updated in the future when changes need to be made. To add a new role or change the permission assignments for a role, change the `roles` dictionary at the top of the function and then run the function again. Note that the `"Anonymous"` role does not need to be represented in the database, as it is the role that represents users who are not known and therefore are not in the database.

Note also that `insert_roles()` is a *static method*, a special type of method that does not require an object to be created as it can be invoked directly on the class, for example, as `Role.insert_roles()`. Static methods do not take a `self` argument like instance methods.

Role Assignment

When users register an account with the application, the correct role should be assigned to them. For most users, the role assigned at registration time will be the `"User"` role, as that is the role that is marked as a default. The only exception is made for the administrator, who needs to be assigned the `"Administrator"` role from the start. This user is identified by an email address stored in the `FLASKY_ADMIN` configuration variable, so as soon as that email address appears in a registration request it can be given the correct role. Example 9-5 shows how this is done in the `User` model constructor.

Example 9-5. app/models.py: defining a default role for users

```
class User(UserMixin, db.Model):
    # ...
    def __init__(self, **kwargs):
        super(User, self).__init__(**kwargs)
        if self.role is None:
            if self.email == current_app.config['FLASKY_ADMIN']:
                self.role = Role.query.filter_by(name='Administrator').first()
            if self.role is None:
                self.role = Role.query.filter_by(default=True).first()
    # ...
```

The `User` constructor first invokes the constructors of the base classes, and if after that the object does not have a role defined, it sets the administrator or default role depending on the email address.

Role Verification

To simplify the implementation of roles and permissions, a helper method can be added to the `User` model that checks whether users have a given permission in the role they have been assigned. The implementation simply defers to the role methods added previously. This is shown in Example 9-6.

Example 9-6. app/models.py: evaluating whether a user has a given permission

```
from flask_login import UserMixin, AnonymousUserMixin

class User(UserMixin, db.Model):
    # ...

    def can(self, perm):
        return self.role is not None and self.role.has_permission(perm)

    def is_administrator(self):
        return self.can(Permission.ADMIN)

class AnonymousUser(AnonymousUserMixin):
    def can(self, permissions):
        return False

    def is_administrator(self):
        return False

login_manager.anonymous_user = AnonymousUser
```

The `can()` method added to the `User` model returns `True` if the requested permission is present in the role, which means that the user should be allowed to perform the requested task. The check for administration permissions is so common that it is also implemented as a standalone `is_administrator()` method.

For added convenience, a custom `AnonymousUser` class that implements the `can()` and `is_administrator()` methods is created as well. This will enable the application to freely call `current_user.can()` and `current_user.is_administrator()` without having to check whether the user is logged in first. Flask-Login is told to use the application's custom anonymous user by setting its class in the `login_manager.anonymous_user` attribute.

For cases in which an entire view function needs to be made available only to users with certain permissions, a custom decorator can be used. Example 9-7 shows the implementation of two decorators, one for generic permission checks and one that checks specifically for the administrator permission.

Example 9-7. app/decorators.py: custom decorators that check user permissions

```
from functools import wraps
from flask import abort
from flask_login import current_user
from .models import Permission

def permission_required(permission):
    def decorator(f):
        @wraps(f)
        def decorated_function(*args, **kwargs):
            if not current_user.can(permission):
                abort(403)
            return f(*args, **kwargs)
        return decorated_function
    return decorator

def admin_required(f):
    return permission_required(Permission.ADMIN)(f)
```

These decorators are built with the help of the `functools` package from the Python standard library and return a 403 response, the "Forbidden" HTTP status code, when the current user does not have the requested permission. In Chapter 3, custom error pages were created for errors 404 and 500, so now a page for the 403 error is added in a similar way.

The following are two examples that demonstrate the usage of these decorators:

```
from .decorators import admin_required, permission_required

@main.route('/admin')
@login_required
@admin_required
def for_admins_only():
    return "For administrators!"

@main.route('/moderate')
@login_required
@permission_required(Permission.MODERATE)
def for_moderators_only():
    return "For comment moderators!"
```

As a rule of thumb, the `route` decorator from Flask should be given first when using multiple decorators in a view function. The remaining decorators should be given in the order in which they need to evaluate when the view function is invoked. In these two cases, the user authenticated state needs to be checked first, since the user needs to be redirected to the login prompt if found to not be authenticated.

Permissions may also need to be checked from templates, so the `Permission` class with all its constants needs to be accessible to them. To avoid having to add a tem-

plate argument in every `render_template()` call, a *context processor* can be used. Context processors make variables available to all templates during rendering. This change is shown in Example 9-8.

Example 9-8. app/main/__init__.py: adding the Permission class to the template context

```
@main.app_context_processor
def inject_permissions():
    return dict(Permission=Permission)
```

The new roles and permissions can be exercised in unit tests. Example 9-9 shows two of the tests. The source code on GitHub includes one for each role.

Example 9-9. tests/test_user_model.py: unit tests for roles and permissions

```
class UserModelTestCase(unittest.TestCase):
    # ...

    def test_user_role(self):
        u = User(email='john@example.com', password='cat')
        self.assertTrue(u.can(Permission.FOLLOW))
        self.assertTrue(u.can(Permission.COMMENT))
        self.assertTrue(u.can(Permission.WRITE))
        self.assertFalse(u.can(Permission.MODERATE))
        self.assertFalse(u.can(Permission.ADMIN))

    def test_anonymous_user(self):
        u = AnonymousUser()
        self.assertFalse(u.can(Permission.FOLLOW))
        self.assertFalse(u.can(Permission.COMMENT))
        self.assertFalse(u.can(Permission.WRITE))
        self.assertFalse(u.can(Permission.MODERATE))
        self.assertFalse(u.can(Permission.ADMIN))
```

> If you have cloned the application's Git repository on GitHub, you can run `git checkout 9a` to check out this version of the application. This update contains a database migration, so remember to run `flask db upgrade` after you check out the code.

Before you move on to the next chapter, add the new roles to your development database in a shell session:

```
(venv) $ flask shell
>>> Role.insert_roles()
>>> Role.query.all()
[<Role 'Administrator'>, <Role 'User'>, <Role 'Moderator'>]
```

It is also a good idea to update the user list so that all the user accounts that were created before roles and permissions existed have a role assigned. You can run the following code in a Python shell to perform this update:

```
(venv) $ flask shell
>>> admin_role = Role.query.filter_by(name='Administrator').first()
>>> default_role = Role.query.filter_by(default=True).first()
>>> for u in User.query.all():
...     if u.role is None:
...         if u.email == app.config['FLASKY_ADMIN']:
...             u.role = admin_role
...         else:
...             u.role = default_role
...
>>> db.session.commit()
```

The user system is now fairly complete. The next chapter will make use of it to create user profile pages.

CHAPTER 10
User Profiles

In this chapter, user profiles for Flasky are implemented. All socially aware sites give their users a profile page, where a summary of the user's participation in the website is presented. Users can advertise their presence on the website by sharing the URL to their profile page, so it is important that the URLs be short and easy to remember.

Profile Information

To make user profile pages more interesting, some additional information about users can be stored in the database. In Example 10-1 the `User` model is extended with several new fields.

Example 10-1. app/models.py: user information fields

```
class User(UserMixin, db.Model):
    # ...
    name = db.Column(db.String(64))
    location = db.Column(db.String(64))
    about_me = db.Column(db.Text())
    member_since = db.Column(db.DateTime(), default=datetime.utcnow)
    last_seen = db.Column(db.DateTime(), default=datetime.utcnow)
```

The new fields store the user's real name, location, self-written bio, date of registration, and date of last visit. The `about_me` field is assigned the type `db.Text()`. The difference between `db.String` and `db.Text` is that `db.Text` is a variable-length field and as such does not need a maximum length.

The two timestamps are given a default value of the current time. Note that the `datetime.utcnow` is missing the `()` at the end. This is because the `default` argument in `db.Column()` can take a function as a value. Each time a default value needs to be

generated, SQLAlchemy invokes the function to produce it. This default value is all that is needed to manage the `member_since` field.

The `last_seen` field is also initialized to the current time upon creation, but it needs to be refreshed each time the user accesses the site. A method in the `User` class can be added to perform this update. This is shown in Example 10-2.

Example 10-2. app/models.py: refreshing a user's last visit time

```
class User(UserMixin, db.Model):
    # ...

    def ping(self):
        self.last_seen = datetime.utcnow()
        db.session.add(self)
        db.session.commit()
```

To keep the last visit date for all users updated, the `ping()` method must be called each time a request from a user is received. Because the `before_app_request` handler in the `auth` blueprint runs before every request, it can do this easily, as shown in Example 10-3.

Example 10-3. app/auth/views.py: pinging the logged-in user

```
@auth.before_app_request
def before_request():
    if current_user.is_authenticated:
        current_user.ping()
        if not current_user.confirmed \
                and request.endpoint \
                and request.blueprint != 'auth' \
                and request.endpoint != 'static':
            return redirect(url_for('auth.unconfirmed'))
```

User Profile Page

Creating a profile page for each user does not present any new challenges. Example 10-4 shows the route definition.

Example 10-4. app/main/views.py: profile page route

```
@main.route('/user/<username>')
def user(username):
    user = User.query.filter_by(username=username).first_or_404()
    return render_template('user.html', user=user)
```

This route is added in the main blueprint. For a user named john, the profile page will be at *http://localhost:5000/user/john*. The username given in the URL is searched in the database and, if found, the *user.html* template is rendered with it as the argument. An invalid username sent into this route will cause a 404 error to be returned. With Flask-SQLAlchemy, the search and error cases can be nicely combined in a single statement using the first_or_404() method of the query object. The *user.html* template is going to need to present user information, so it receives the user object as an argument. An initial version of this template is shown in Example 10-5.

Example 10-5. app/templates/user.html: user profile template

```
{% extends "base.html" %}
{% block title %}Flasky - {{ user.username }}{% endblock %}

{% block page_content %}
<div class="page-header">
    <h1>{{ user.username }}</h1>
    {% if user.name or user.location %}
    <p>
        {% if user.name %}{{ user.name }}{% endif %}
        {% if user.location %}
            From <a href="http://maps.google.com/?q={{ user.location }}">
                {{ user.location }}
            </a>
        {% endif %}
    </p>
    {% endif %}
    {% if current_user.is_administrator() %}
    <p><a href="mailto:{{ user.email }}">{{ user.email }}</a></p>
    {% endif %}
    {% if user.about_me %}<p>{{ user.about_me }}</p>{% endif %}
    <p>
        Member since {{ moment(user.member_since).format('L') }}.
        Last seen {{ moment(user.last_seen).fromNow() }}.
    </p>
</div>
{% endblock %}
```

This template has a few interesting implementation details:

- The name and location fields are rendered inside a single <p> element. A Jinja2 conditional ensures that the <p> element is created only when at least one of the fields is defined.
- The user location field is rendered as a link to a Google Maps query, so that clicking on it opens a map centered on the location.

- If the logged-in user is an administrator, then the email address of the user is shown, rendered as a *mailto* link. This is useful when an administrator is viewing the profile page of another user and needs to contact the user.
- The two timestamps for the user are rendered to the page using Flask-Moment, as shown in Chapter 3.

As most users will want easy access to their own profile page, a link to it can be added to the navigation bar. The relevant changes to the *base.html* template are shown in Example 10-6.

Example 10-6. app/templates/base.html: add link to profile page in the navigation bar

```
{% if current_user.is_authenticated %}
<li>
    <a href="{{ url_for('main.user', username=current_user.username) }}">
        Profile
    </a>
</li>
{% endif %}
```

Using a conditional for the profile page link is necessary because the navigation bar is also rendered for unauthenticated users, in which case the profile link is skipped. Figure 10-1 shows how the profile page looks in the browser. The new profile link in the navigation bar is also shown.

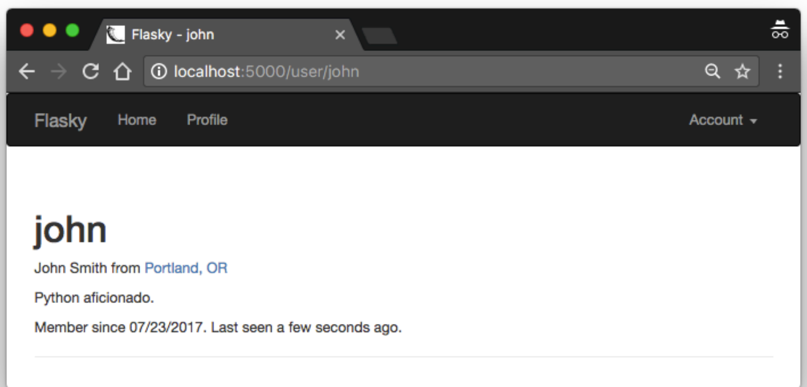


Figure 10-1. User profile page

If you have cloned the application's Git repository on GitHub, you can run `git checkout 10a` to check out this version of the application. This update contains a database migration, so remember to run `flask db upgrade` after you check out the code.

Profile Editor

There are two different use cases related to editing of user profiles. The most obvious is that users need to have access to a page where they can enter information about themselves to present in their profile pages. A less obvious but also important requirement is to let administrators edit the profiles of other users—not only their personal information items but also other fields in the `User` model to which users have no direct access, such as the user role. Because the two profile editing requirements are substantially different, two different forms will be created.

User-Level Profile Editor

The profile editing form for regular users is shown in Example 10-7.

Example 10-7. app/main/forms.py: edit profile form

```
class EditProfileForm(FlaskForm):
    name = StringField('Real name', validators=[Length(0, 64)])
    location = StringField('Location', validators=[Length(0, 64)])
    about_me = TextAreaField('About me')
    submit = SubmitField('Submit')
```

Note that as all the fields in this form are optional, the length validator allows a length of zero as a minimum. The route definition that uses this form is shown in Example 10-8.

Example 10-8. app/main/views.py: edit profile route

```
@main.route('/edit-profile', methods=['GET', 'POST'])
@login_required
def edit_profile():
    form = EditProfileForm()
    if form.validate_on_submit():
        current_user.name = form.name.data
        current_user.location = form.location.data
        current_user.about_me = form.about_me.data
        db.session.add(current_user._get_current_object())
        db.session.commit()
        flash('Your profile has been updated.')
        return redirect(url_for('.user', username=current_user.username))
    form.name.data = current_user.name
```

```
    form.location.data = current_user.location
    form.about_me.data = current_user.about_me
    return render_template('edit_profile.html', form=form)
```

As in previous forms, the data associated with each form field is available at `form.<field-name>.data`. This is useful not only to obtain values submitted by the user, but also to provide initial values that are shown to the user for editing. When `form.validate_on_submit()` is `False`, the three fields in this form are initialized from the corresponding fields in `current_user`. Then, when the form is submitted, the `data` attributes of the form fields contain the updated values, so these are moved back into the fields of the user object before the object is saved back to the database. Figure 10-2 shows the profile editing page.

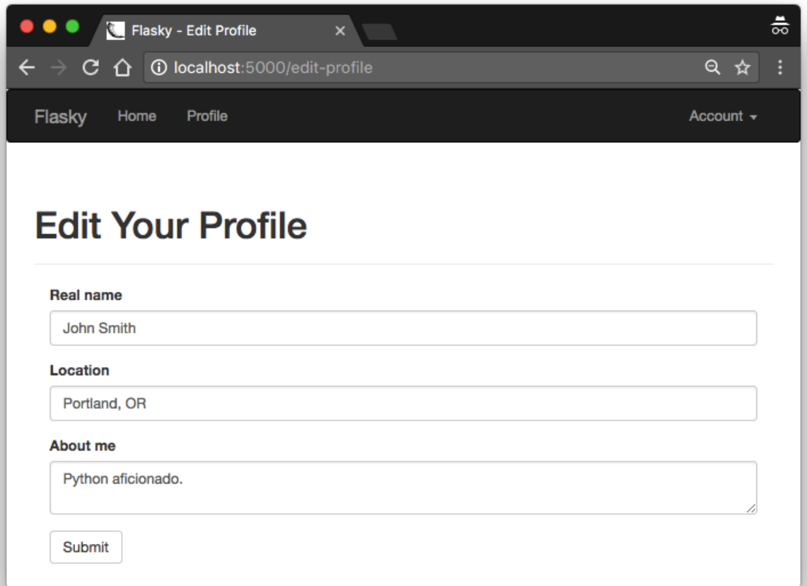


Figure 10-2. Profile editor

To make it easy for users to reach this page, a direct link can be added in the profile page, as shown in Example 10-9.

Example 10-9. app/templates/user.html: edit profile link

```
{% if user == current_user %}
<a class="btn btn-default" href="{{ url_for('.edit_profile') }}">
    Edit Profile
</a>
{% endif %}
```

The conditional that encloses the link will make the link appear only when users are viewing their own profiles.

Administrator-Level Profile Editor

The profile editing form for administrators is more complex than the one for regular users. In addition to the three profile information fields, this form allows administrators to edit a user's email, username, confirmed status, and role. The form is shown in Example 10-10.

Example 10-10. app/main/forms.py: profile editing form for administrators

```
class EditProfileAdminForm(FlaskForm):
    email = StringField('Email', validators=[DataRequired(), Length(1, 64),
                                             Email()])
    username = StringField('Username', validators=[
        DataRequired(), Length(1, 64),
        Regexp('^[A-Za-z][A-Za-z0-9_.]*$', 0,
               'Usernames must have only letters, numbers, dots or '
               'underscores')])
    confirmed = BooleanField('Confirmed')
    role = SelectField('Role', coerce=int)
    name = StringField('Real name', validators=[Length(0, 64)])
    location = StringField('Location', validators=[Length(0, 64)])
    about_me = TextAreaField('About me')
    submit = SubmitField('Submit')

    def __init__(self, user, *args, **kwargs):
        super(EditProfileAdminForm, self).__init__(*args, **kwargs)
        self.role.choices = [(role.id, role.name)
                             for role in Role.query.order_by(Role.name).all()]
        self.user = user

    def validate_email(self, field):
        if field.data != self.user.email and \
                User.query.filter_by(email=field.data).first():
            raise ValidationError('Email already registered.')

    def validate_username(self, field):
        if field.data != self.user.username and \
```

```
            User.query.filter_by(username=field.data).first():
        raise ValidationError('Username already in use.')
```

The `SelectField` is WTForm's wrapper for the `<select>` HTML form control, which implements a drop-down list, used in this form to select a user role. An instance of `SelectField` must have the items set in its `choices` attribute. They must be given as a list of tuples, with each tuple consisting of two values: an identifier for the item and the text to show in the control as a string. The `choices` list is set in the form's constructor, with values obtained from the `Role` model with a query that sorts all the roles alphabetically by name. The identifier for each tuple is set to the `id` of each role, and since these are integers, a `coerce=int` argument is added to the `SelectField` constructor so that the field values are stored as integers instead of the default, which is strings.

The `email` and `username` fields are constructed in the same way as in the authentication forms, but their validation requires some careful handling. The validation condition used for both these fields must first check whether a change to the field was made, and only when there is a change should it ensure that the new value does not duplicate another user's. When these fields are not changed, then validation should pass. To implement this logic, the form's constructor receives the user object as an argument and saves it as a member variable, which is later used in the custom validation methods.

The route definition for the administrator's profile editor is shown in Example 10-11.

Example 10-11. app/main/views.py: edit profile route for administrators

```
from ..decorators import admin_required

@main.route('/edit-profile/<int:id>', methods=['GET', 'POST'])
@login_required
@admin_required
def edit_profile_admin(id):
    user = User.query.get_or_404(id)
    form = EditProfileAdminForm(user=user)
    if form.validate_on_submit():
        user.email = form.email.data
        user.username = form.username.data
        user.confirmed = form.confirmed.data
        user.role = Role.query.get(form.role.data)
        user.name = form.name.data
        user.location = form.location.data
        user.about_me = form.about_me.data
        db.session.add(user)
        db.session.commit()
        flash('The profile has been updated.')
        return redirect(url_for('.user', username=user.username))
    form.email.data = user.email
```

```
    form.username.data = user.username
    form.confirmed.data = user.confirmed
    form.role.data = user.role_id
    form.name.data = user.name
    form.location.data = user.location
    form.about_me.data = user.about_me
    return render_template('edit_profile.html', form=form, user=user)
```

This route has largely the same structure as the simpler one for regular users, but it includes the `admin_required` decorator created in Chapter 9, which will automatically return a 403 error for any users who are not administrators that try to use this route.

The user `id` is given as a dynamic argument in the URL, so Flask-SQLAlchemy's `get_or_404()` convenience function can be used, knowing that if the `id` is invalid the request will return a code 404 error. The `SelectField` used for the user role also deserves to be studied. When setting the initial value for the field, the `role_id` is assigned to `field.role.data` because the list of tuples set in the `choices` attribute uses the numeric identifiers to reference each option. When the form is submitted, the `id` is extracted from the field's `data` attribute and used in a query to load the selected role object by its `id` once again. The `coerce=int` argument used in the `SelectField` declaration in the form ensures that the `data` attribute of this field is always converted to an integer.

To link to this page, another button is added in the user profile page, as shown in Example 10-12.

Example 10-12. app/templates/user.html: profile editing link for administrators

```
{% if current_user.is_administrator() %}
<a class="btn btn-danger"
        href="{{ url_for('.edit_profile_admin', id=user.id) }}">
    Edit Profile [Admin]
</a>
{% endif %}
```

This button is rendered with a different Bootstrap style to call attention to it. The conditional that wraps it makes the button appear in profile pages only if the logged-in user has the administrator role.

> If you have cloned the application's Git repository on GitHub, you can run `git checkout 10b` to check out this version of the application.

User Avatars

The look of the profile pages can be improved by showing avatar pictures of users. In this section, you will learn how to add user avatars provided by Gravatar, the leading avatar service. Gravatar associates avatar images with email addresses. Users create an account at *https://gravatar.com* and then upload their images. The service exposes the user's avatar through a specially crafted URL that includes the MD5 hash of the user's email address, which can be calculated as follows:

```
(venv) $ python
>>> import hashlib
>>> hashlib.md5('john@example.com'.encode('utf-8')).hexdigest()
'd4c74594d841139328695756648b6bd6'
```

The avatar URLs are then generated by appending the MD5 hash to the *https://secure.gravatar.com/avatar/* URL. For example, you can type *https://secure.gravatar.com/avatar/d4c74594d841139328695756648b6bd6* in your browser's address bar to get the avatar image for the email address *john@example.com*, or a default avatar image if that email address does not have an avatar registered. After you build the basic avatar URL, a few query string arguments can be used to configure the characteristics of the avatar image, as described in Table 10-1.

Table 10-1. Gravatar query string arguments

Argument name	Description
s	Image size, in pixels.
r	Image rating. Options are "g", "pg", "r", and "x".
d	The default image generator for users who have no avatars registered with the Gravatar service. Options are "404" to return a 404 error, a URL that points to a default image, or one of the following image generators: "mm", "identicon", "monsterid", "wavatar", "retro", or "blank".
fd	Force the use of default avatars.

For example, adding *?d=identicon* to the avatar URL for *john@example.com* will generate a different default avatar that is based on geometric designs. All these options to generate avatar URLs can be added to the `User` model. The implementation is shown in Example 10-13.

Example 10-13. app/models.py: gravatar URL generation

```
import hashlib
from flask import request

class User(UserMixin, db.Model):
    # ...
    def gravatar(self, size=100, default='identicon', rating='g'):
```

```
        url = 'https://secure.gravatar.com/avatar'
        hash = hashlib.md5(self.email.lower().encode('utf-8')).hexdigest()
        return '{url}/{hash}?s={size}&d={default}&r={rating}'.format(
            url=url, hash=hash, size=size, default=default, rating=rating)
```

The avatar URL is generated from the base URL, the MD5 hash of the user's email address, and the arguments, all of which have default values. Note that one of the requirements of the Gravatar service is that the email address from which the MD5 hash is obtained is normalized to contain only lowercase alphabetical characters, so that conversion is also added to this method. With this implementation it is easy to generate avatar URLs in the Python shell:

```
(venv) $ flask shell
>>> u = User(email='john@example.com')
>>> u.gravatar()
'https://secure.gravatar.com/avatar/d4c74594d841139328695756648b6bd6?s=100&d=
identicon&r=g'
>>> u.gravatar(size=256)
'https://secure.gravatar.com/avatar/d4c74594d841139328695756648b6bd6?s=256&d=
identicon&r=g'
```

The `gravatar()` method can also be invoked from Jinja2 templates. Example 10-14 shows how a 256-pixel avatar can be added to the profile page.

Example 10-14. app/templates/user.html: adding an avatar to the profile page

```
...
<img class="img-rounded profile-thumbnail" src="{{ user.gravatar(size=256) }}">
<div class="profile-header">
   ...
</div>
...
```

The `profile-thumbnail` CSS class helps with the positioning of the image on the page. The `<div>` element that follows the image encapsulates the profile information and uses the `profile-header` CSS class to improve the formatting. You can see the definition of the CSS class in the GitHub repository for the application.

Using a similar approach, the base template adds a small thumbnail image of the logged-in user in the navigation bar. To better format the avatar pictures in the page, custom CSS classes are used. You can find these in the source code repository in a *styles.css* file added to the application's static file folder and referenced from the *base.html* template. Figure 10-3 shows the user profile page with avatar.

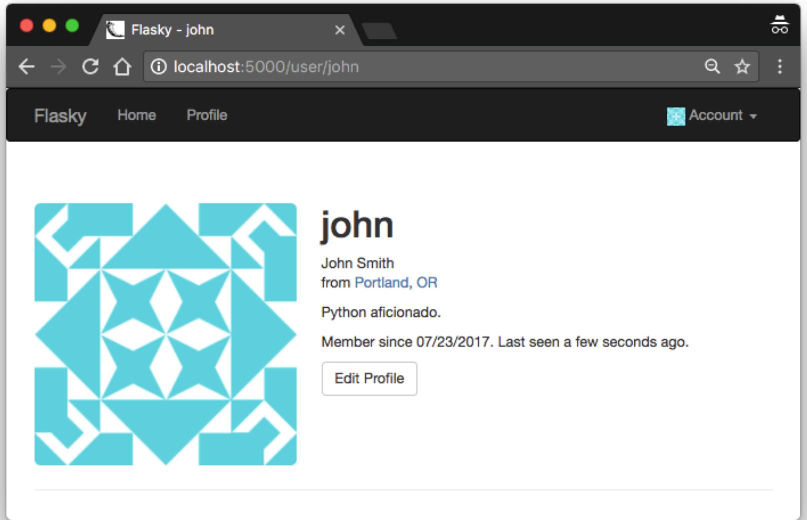


Figure 10-3. User profile page with avatar

> If you have cloned the application's Git repository on GitHub, you can run `git checkout 10c` to check out this version of the application.

The generation of avatars requires an MD5 hash to be generated, which is a CPU-intensive operation. If a large number of avatars need to be generated for a page, then the computational work can add up and become significant. Since the MD5 hash for a user will remain constant for as long as the email address stays the same, it can be *cached* in the `User` model. Example 10-15 shows the changes to the `User` model to store the MD5 hashes in the database.

Example 10-15. app/models.py: gravatar URL generation with caching of MD5 hashes

```
class User(UserMixin, db.Model):
    # ...
    avatar_hash = db.Column(db.String(32))

    def __init__(self, **kwargs):
        # ...
        if self.email is not None and self.avatar_hash is None:
```

```
            self.avatar_hash = self.gravatar_hash()

    def change_email(self, token):
        # ...
        self.email = new_email
        self.avatar_hash = self.gravatar_hash()
        db.session.add(self)
        return True

    def gravatar_hash(self):
        return hashlib.md5(self.email.lower().encode('utf-8')).hexdigest()

    def gravatar(self, size=100, default='identicon', rating='g'):
        if request.is_secure:
            url = 'https://secure.gravatar.com/avatar'
        else:
            url = 'http://www.gravatar.com/avatar'
        hash = self.avatar_hash or self.gravatar_hash()
        return '{url}/{hash}?s={size}&d={default}&r={rating}'.format(
            url=url, hash=hash, size=size, default=default, rating=rating)
```

To avoid duplicating the logic to compute the gravatar hash, a new method, `gravatar_hash()`, is added that performs this task. During model initialization, the hash is stored in the new `avatar_hash` model column. If the user updates the email address, then the hash is recalculated. The `gravatar()` method uses the stored hash if available, and if not, it generates a new hash as before.

If you have cloned the application's Git repository on GitHub, you can run `git checkout 10d` to check out this version of the application. This update contains a database migration, so remember to run `flask db upgrade` after you check out the code.

In the next chapter, the blogging engine that powers this application will be created.

CHAPTER 11

Blog Posts

This chapter is dedicated to the implementation of Flasky's main feature, which is to allow users to read and write blog posts. Here you will learn a few new techniques for reuse of templates, pagination of long lists of items, and working with rich text.

Blog Post Submission and Display

To support blog posts, a new database model that represents them is necessary. This model is shown in Example 11-1.

Example 11-1. app/models.py: Post model

```
class Post(db.Model):
    __tablename__ = 'posts'
    id = db.Column(db.Integer, primary_key=True)
    body = db.Column(db.Text)
    timestamp = db.Column(db.DateTime, index=True, default=datetime.utcnow)
    author_id = db.Column(db.Integer, db.ForeignKey('users.id'))

class User(UserMixin, db.Model):
    # ...
    posts = db.relationship('Post', backref='author', lazy='dynamic')
```

A blog post is represented by a body, a timestamp, and a one-to-many relationship from the `User` model. The `body` field is defined with type `db.Text` so that there is no limitation on the length.

The form that will be shown in the main page of the application lets users write a blog post. This form is very simple; it contains just a text area where the blog post can be typed and a submit button. The form definition is shown in Example 11-2.

Example 11-2. app/main/forms.py: blog post form

```
class PostForm(FlaskForm):
    body = TextAreaField("What's on your mind?", validators=[DataRequired()])
    submit = SubmitField('Submit')
```

The `index()` view function handles the form and passes the list of old blog posts to the template, as shown in Example 11-3.

Example 11-3. app/main/views.py: home page route with a blog post

```
@main.route('/', methods=['GET', 'POST'])
def index():
    form = PostForm()
    if current_user.can(Permission.WRITE_ARTICLES) and form.validate_on_submit():
        post = Post(body=form.body.data,
                    author=current_user._get_current_object())
        db.session.add(post)
        db.session.commit()
        return redirect(url_for('.index'))
    posts = Post.query.order_by(Post.timestamp.desc()).all()
    return render_template('index.html', form=form, posts=posts)
```

This view function passes the form and the complete list of blog posts to the template. The list of posts is ordered by timestamp, in descending order. The blog post form is handled in the usual manner, with the creation of a new `Post` instance when a valid submission is received. The current user's permission to write articles is checked before allowing the new post.

Note how the `author` attribute of the new post object is set to the expression `current_user._get_current_object()`. The `current_user` variable from Flask-Login, like all context variables, is implemented as a thread-local proxy object. This object behaves like a user object but is really a thin wrapper that contains the actual user object inside. The database needs a real user object, which is obtained by calling `_get_current_object()` on the proxy object.

The form is rendered below the greeting in the *index.html* template, followed by the blog posts. The list of blog posts is a first attempt to create a blog post timeline, with all the blog posts in the database listed in chronological order from newest to oldest. The changes to the template are shown in Example 11-4.

Example 11-4. app/templates/index.html: home page template with blog posts

```
{% extends "base.html" %}
{% import "bootstrap/wtf.html" as wtf %}
...
<div>
    {% if current_user.can(Permission.WRITE_ARTICLES) %}
    {{ wtf.quick_form(form) }}
    {% endif %}
</div>
<ul class="posts">
    {% for post in posts %}
    <li class="post">
        <div class="profile-thumbnail">
            <a href="{{ url_for('.user', username=post.author.username) }}">
                <img class="img-rounded profile-thumbnail"
                    src="{{ post.author.gravatar(size=40) }}">
            </a>
        </div>
        <div class="post-date">{{ moment(post.timestamp).fromNow() }}</div>
        <div class="post-author">
            <a href="{{ url_for('.user', username=post.author.username) }}">
                {{ post.author.username }}
            </a>
        </div>
        <div class="post-body">{{ post.body }}</div>
    </li>
    {% endfor %}
</ul>
...
```

Note that the `User.can()` method is used to skip the blog post form for users who do not have the `WRITE` permission in their role. The blog post list is implemented as an HTML unordered list, with CSS classes giving it a nicer formatting. A small avatar of the author is rendered on the left side, and both the avatar and the author's username are rendered as links to the user's profile page. The CSS styles used are stored in the *styles.css* file located in the application's *static* directory. You can review this file in the GitHub repository. Figure 11-1 shows the home page with submission form and blog post list.

If you have cloned the application's Git repository on GitHub, you can run `git checkout 11a` to check out this version of the application. This update contains a database migration, so remember to run `flask db upgrade` after you check out the code.

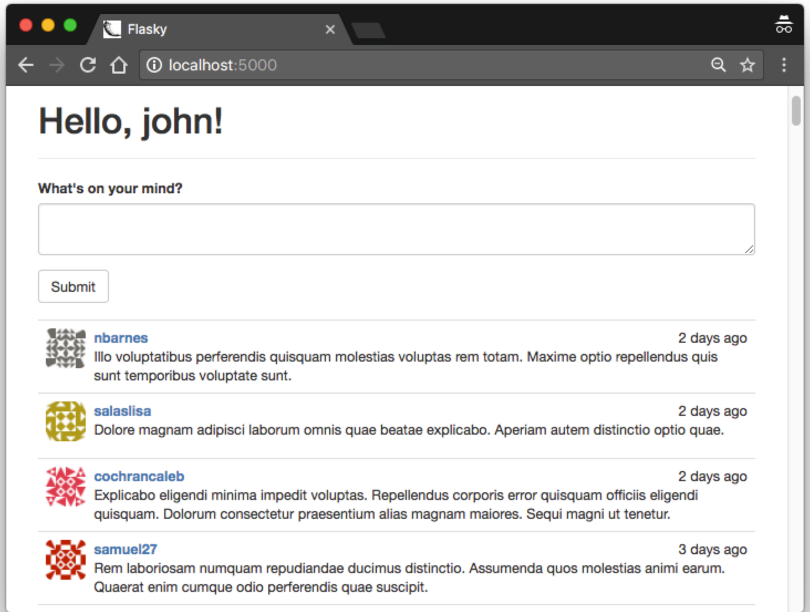


Figure 11-1. Home page with blog submission form and blog post list

Blog Posts on Profile Pages

The user profile page can be improved by showing a list of blog posts authored by the user. Example 11-5 shows the changes to the view function to obtain the post list.

Example 11-5. app/main/views.py: profile page route with blog posts

```
@main.route('/user/<username>')
def user(username):
    user = User.query.filter_by(username=username).first()
    if user is None:
        abort(404)
    posts = user.posts.order_by(Post.timestamp.desc()).all()
    return render_template('user.html', user=user, posts=posts)
```

The list of blog posts for a user is obtained from the `User.posts` relationship. This works like a query object, so filters such as `order_by()` can be used on it like in a regular query object.

The *user.html* template needs to have the same `<ul>` HTML tree that renders a list of blog posts in *index.html*, but having to maintain two identical copies of a piece of HTML code is not ideal. For cases like this, Jinja2's `include` directive is very useful. The snippet of HTML that generates the post list can be moved to a separate file that both *index.html* and *user.html* can include. Example 11-6 shows how this include looks in *user.html*.

Example 11-6. app/templates/user.html: profile page template with blog posts

```
...
<h3>Posts by {{ user.username }}</h3>
{% include '_posts.html' %}
...
```

To complete this reorganization, the `<ul>` tree from *index.html* is moved to the new template *_posts.html*, and replaced with another `include` directive like the one just shown. Note that the use of an underscore prefix in the *_posts.html* template name is not a requirement; this is merely a convention to distinguish full and partial templates.

> If you have cloned the application's Git repository on GitHub, you can run `git checkout 11b` to check out this version of the application.

Paginating Long Blog Post Lists

As the site grows and the number of blog posts increases, it will become slow and impractical to show the complete list of posts on the home and profile pages. Big pages take longer to generate, download, and render in the web browser, so the quality of the user experience decreases as the pages get larger. The solution is to *paginate* the data and render it in chunks.

Creating Fake Blog Post Data

To be able to work with multiple pages of blog posts, it is necessary to have a test database with a large volume of data. Manually adding new database entries is time consuming and tedious; an automated solution is more appropriate. There are several Python packages that can be used to generate fake information. A fairly complete one is *Faker*, which is installed with *pip*:

```
(venv) $ pip install faker
```

The Faker package is not, strictly speaking, a dependency of the application, because it is needed only during development. To separate the production dependencies from the development dependencies, the *requirements.txt* file can be replaced with a *requirements* subdirectory that stores different sets of dependencies. Inside this new subdirectory, a *dev.txt* file can list the dependencies that are necessary for development and a *prod.txt* file can list the dependencies that are needed in production. As there are a large number of dependencies that will be in both lists, a *common.txt* file is added for those, and then the *dev.txt* and *prod.txt* lists use the `-r` prefix to include it. Example 11-7 shows the *dev.txt* file.

Example 11-7. requirements/dev.txt: development requirements file

```
-r common.txt
faker==0.7.18
```

Example 11-8 shows a new module added to the application that contains two functions that generate fake users and posts.

Example 11-8. app/fake.py: generating fake users and blog posts

```
from random import randint
from sqlalchemy.exc import IntegrityError
from faker import Faker
from . import db
from .models import User, Post

def users(count=100):
    fake = Faker()
    i = 0
    while i < count:
        u = User(email=fake.email(),
                 username=fake.user_name(),
                 password='password',
                 confirmed=True,
                 name=fake.name(),
                 location=fake.city(),
                 about_me=fake.text(),
                 member_since=fake.past_date())
        db.session.add(u)
        try:
            db.session.commit()
            i += 1
        except IntegrityError:
            db.session.rollback()

def posts(count=100):
    fake = Faker()
    user_count = User.query.count()
```

```
    for i in range(count):
        u = User.query.offset(randint(0, user_count - 1)).first()
        p = Post(body=fake.text(),
                 timestamp=fake.past_date(),
                 author=u)
        db.session.add(p)
    db.session.commit()
```

The attributes of these fake objects are produced by random information generators provided by the Faker package, which can generate real-looking names, emails, sentences, and many more attributes.

The email addresses and usernames of users must be unique, but since Faker generates these in a completely random fashion, there is a risk of having duplicates. In the unlikely event that a duplicate is generated, the database session commit will throw an `IntegrityError` exception. The exception is handled by rolling back the session to cancel that duplicate user. The loop will run until the requested number of unique users are generated.

The random post generation must assign a random user to each post. For this, the `offset()` query filter is used. This filter discards the number of results given as an argument. By setting a random offset and then calling `first()`, a different random user is obtained each time.

> If you have cloned the application's Git repository on GitHub, you can run `git checkout 11c` to check out this version of the application. To ensure that you have all the dependencies installed, also run `pip install -r requirements/dev.txt`.

The new functions make it easy to create a large number of fake users and posts from the Python shell:

```
(venv) $ flask shell
>>> from app import fake
>>> fake.users(100)
>>> fake.posts(100)
```

If you run the application now, you will see a long list of random blog posts on the home page, by many different users.

Rendering in Pages

Example 11-9 shows the changes to the home page route to support pagination.

Example 11-9. app/main/views.py: paginating the blog post list

```
@main.route('/', methods=['GET', 'POST'])
def index():
    # ...
    page = request.args.get('page', 1, type=int)
    pagination = Post.query.order_by(Post.timestamp.desc()).paginate(
        page, per_page=current_app.config['FLASKY_POSTS_PER_PAGE'],
        error_out=False)
    posts = pagination.items
    return render_template('index.html', form=form, posts=posts,
                           pagination=pagination)
```

The page number to render is obtained from the request's query string, which is available as `request.args`. When a page isn't given, a default page of 1 (the first page) is used. The `type=int` argument ensures that if the argument cannot be converted to an integer, the default value is returned.

To load a single page of records, the final call to the `all()` method of the query object is replaced with Flask-SQLAlchemy's `paginate()`. The `paginate()` method takes the page number as its first and only required argument. An optional `per_page` argument can be given to indicate the size of each page, in number of items. If this argument is not specified, the default is 20 items per page. Another optional argument called `error_out` can be set to `True` (the default) to issue a code 404 error when a page outside of the valid range is requested. If `error_out` is `False`, pages outside of the valid range are returned with an empty list of items. To make the page sizes configurable, the value of the `per_page` argument is read from an application-specific configuration variable called `FLASKY_POSTS_PER_PAGE` that is added in *config.py*.

With these changes, the blog post list on the home page will show a limited number of items. To see the second page of posts, add a *?page=2* query string to the URL in the browser's address bar.

Adding a Pagination Widget

The return value of `paginate()` is an object of class `Pagination`, a class defined by Flask-SQLAlchemy. This object contains several properties that are useful to generate page links in a template, so it is passed to the template as an argument. A summary of the attributes of the pagination object is shown in Table 11-1.

Table 11-1. Flask-SQLAlchemy pagination object attributes

Attribute	Description
`items`	The records in the current page
`query`	The source query that was paginated
`page`	The current page number
`prev_num`	The previous page number
`next_num`	The next page number
`has_next`	`True` if there is a next page
`has_prev`	`True` if there is a previous page
`pages`	The total number of pages for the query
`per_page`	The number of items per page
`total`	The total number of items returned by the query

The pagination object also has some methods, listed in Table 11-2.

Table 11-2. Flask-SQLAlchemy pagination object methods

Method	Description
`iter_pages(left_edge=2, left_current=2, right_current=5, right_edge=2)`	An iterator that returns the sequence of page numbers to display in a pagination widget. The list will have `left_edge` pages on the left side, `left_current` pages to the left of the current page, `right_current` pages to the right of the current page, and `right_edge` pages on the right side. For example, for page 50 of 100 this iterator configured with default values will return the following pages: 1, 2, None, 48, 49, 50, 51, 52, 53, 54, 55, None, 99, 100. A None value in the sequence indicates a gap in the sequence of pages.
`prev()`	A pagination object for the previous page.
`next()`	A pagination object for the next page.

Armed with this powerful object and Bootstrap's pagination CSS classes, it is quite easy to build a pagination footer in the template. The implementation shown in Example 11-10 is done as a reusable Jinja2 macro.

Example 11-10. app/templates/_macros.html: pagination template macro

```
{% macro pagination_widget(pagination, endpoint) %}
<ul class="pagination">
    <li{% if not pagination.has_prev %} class="disabled"{% endif %}>
        <a href="{% if pagination.has_prev %}{{ url_for(endpoint,
            page = pagination.page - 1, **kwargs) }}{% else %}#{% endif %}">
            &laquo;
        </a>
    </li>
    {% for p in pagination.iter_pages() %}
```

```
        {% if p %}
            {% if p == pagination.page %}
            <li class="active">
                <a href="{{ url_for(endpoint, page = p, **kwargs) }}">{{ p }}</a>
            </li>
            {% else %}
            <li>
                <a href="{{ url_for(endpoint, page = p, **kwargs) }}">{{ p }}</a>
            </li>
            {% endif %}
        {% else %}
        <li class="disabled"><a href="#">&hellip;</a></li>
        {% endif %}
    {% endfor %}
    <li{% if not pagination.has_next %} class="disabled"{% endif %}>
        <a href="{% if pagination.has_next %}{{ url_for(endpoint,
            page = pagination.page + 1, **kwargs) }}{% else %}#{% endif %}">
            &raquo;
        </a>
    </li>
</ul>
{% endmacro %}
```

The macro creates a Bootstrap pagination element, which is a styled unordered list. It defines the following page links inside it:

- A "previous page" link. This link gets the `disabled` CSS class if the current page is the first page.
- Links to all pages returned by the pagination object's `iter_pages()` iterator. These pages are rendered as links with an explicit page number, given as an argument to `url_for()`. The page currently displayed is highlighted using the `active` CSS class. Gaps in the sequence of pages are rendered with the ellipsis character.
- A "next page" link. This link will appear disabled if the current page is the last page.

Jinja2 macros always receive keyword arguments without having to include `**kwargs` in the argument list. The pagination macro passes all the keyword arguments it receives to the `url_for()` call that generates the pagination links. This approach can be used with routes such as the profile page that have a dynamic part.

The `pagination_widget` macro can be added below the *_posts.html* template included by *index.html* and *user.html*. Example 11-11 shows how it is used in the application's home page.

Example 11-11. app/templates/index.html: pagination footer for blog post lists

```
{% extends "base.html" %}
{% import "bootstrap/wtf.html" as wtf %}
{% import "_macros.html" as macros %}
...
{% include '_posts.html' %}
<div class="pagination">
    {{ macros.pagination_widget(pagination, '.index') }}
</div>
{% endif %}
```

Figure 11-2 shows how the pagination links appear in the page.



Figure 11-2. Blog post pagination

If you have cloned the application's Git repository on GitHub, you can run `git checkout 11d` to check out this version of the application.

Rich-Text Posts with Markdown and Flask-PageDown

Plain-text posts are sufficient for short messages and status updates, but users who want to write longer articles will find the lack of formatting very limiting. In this section, the text area field where posts are entered will be upgraded to support the Markdown syntax and present a rich-text preview of the post.

The implementation of this feature requires a few new packages:

- PageDown, a client-side Markdown-to-HTML converter implemented in Java-Script
- Flask-PageDown, a PageDown wrapper for Flask that integrates PageDown with Flask-WTF forms
- Markdown, a server-side Markdown-to-HTML converter implemented in Python
- Bleach, an HTML sanitizer implemented in Python

The Python packages can all be installed with *pip*:

```
(venv) $ pip install flask-pagedown markdown bleach
```

Using Flask-PageDown

The Flask-PageDown extension defines a `PageDownField` class that has the same interface as the `TextAreaField` from WTForms. Before this field can be used, the extension needs to be initialized as shown in Example 11-12.

Example 11-12. app/__init__.py: Flask-PageDown initialization

```python
from flask_pagedown import PageDown
# ...
pagedown = PageDown()
# ...
def create_app(config_name):
    # ...
    pagedown.init_app(app)
    # ...
```

To convert the text area control in the home page to a Markdown rich-text editor, the `body` field of the `PostForm` must be changed to a `PageDownField` as shown in Example 11-13.

Example 11-13. app/main/forms.py: Markdown-enabled post form

```python
from flask_pagedown.fields import PageDownField

class PostForm(FlaskForm):
    body = PageDownField("What's on your mind?", validators=[Required()])
    submit = SubmitField('Submit')
```

The Markdown preview is generated with the help of the PageDown libraries, so these must be added to the template. Flask-PageDown simplifies this task by provid-

ing a template macro that includes the required files from a CDN as shown in Example 11-14.

Example 11-14. app/templates/index.html: Flask-PageDown template declaration

```
{% block scripts %}
{{ super() }}
{{ pagedown.include_pagedown() }}
{% endblock %}
```

If you have cloned the application's Git repository on GitHub, you can run `git checkout 11e` to check out this version of the application. To ensure that you have all the dependencies installed also run `pip install -r requirements/dev.txt`.

With these changes, Markdown-formatted text typed in the text area field will be immediately rendered as HTML in the preview area below. Figure 11-3 shows the blog submission form with rich text.

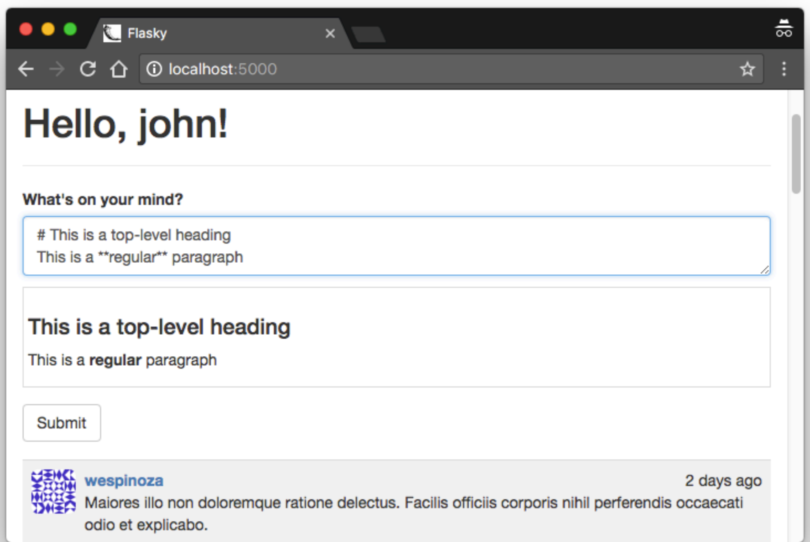


Figure 11-3. Rich-text blog post form

Handling Rich Text on the Server

When the form is submitted, only the raw Markdown text is sent with the POST request; the HTML preview that is shown on the page is discarded. Sending the generated HTML preview with the form can be considered a security risk, as it would be fairly easy for an attacker to construct HTML sequences that do not match the Markdown source and submit them. To avoid any risks, only the Markdown source text is submitted, and once in the server it is converted again to HTML using Markdown, a Python Markdown-to-HTML converter. The resulting HTML is sanitized with Bleach to ensure that only a short list of allowed HTML tags are used.

The conversion of the Markdown blog posts to HTML can be done in the *_posts.html* template, but this is inefficient as posts will have to be converted every time they are rendered to a page. To avoid this repetition, the conversion can be done once when the blog post is created and then cached in the database. The HTML code for the rendered blog post is cached in a new field added to the `Post` model that the template can access directly. The original Markdown source is also kept in the database in case the post needs to be edited. Example 11-15 shows the changes to the `Post` model.

Example 11-15. app/models.py: Markdown text handling in the Post model

```
from markdown import markdown
import bleach

class Post(db.Model):
    # ...
    body_html = db.Column(db.Text)
    # ...
    @staticmethod
    def on_changed_body(target, value, oldvalue, initiator):
        allowed_tags = ['a', 'abbr', 'acronym', 'b', 'blockquote', 'code',
                        'em', 'i', 'li', 'ol', 'pre', 'strong', 'ul',
                        'h1', 'h2', 'h3', 'p']
        target.body_html = bleach.linkify(bleach.clean(
            markdown(value, output_format='html'),
            tags=allowed_tags, strip=True))

db.event.listen(Post.body, 'set', Post.on_changed_body)
```

The `on_changed_body()` function is registered as a listener of SQLAlchemy's "set" event for `body`, which means that it will be automatically invoked whenever the `body` field is set to a new value. The handler function renders the HTML version of the body and stores it in `body_html`, effectively making the conversion of the Markdown text to HTML fully automatic.

The actual conversion is done in three steps. First, the `markdown()` function does an initial conversion to HTML. The result is passed to `clean()`, along with the list of

approved HTML tags. The clean() function removes any tags not on the whitelist. The final conversion is done with linkify(), another function provided by Bleach that converts any URLs written in plain text into proper <a> links. This last step is necessary because automatic link generation is not officially in the Markdown specification, but is a very convenient feature. On the client side, PageDown supports this feature as an optional extension, so linkify() matches that functionality on the server.

The last change is to replace post.body with post.body_html in the template when available, as shown in Example 11-16.

Example 11-16. app/templates/_posts.html: use the HTML version of the post bodies in the template

```
...
<div class="post-body">
    {% if post.body_html %}
        {{ post.body_html | safe }}
    {% else %}
        {{ post.body }}
    {% endif %}
</div>
...
```

The | safe suffix when rendering the HTML body is there to tell Jinja2 not to escape the HTML elements. Jinja2 escapes all template variables by default as a security measure, but the Markdown-generated HTML was generated by the server, so it is safe to render directly as HTML.

> If you have cloned the application's Git repository on GitHub, you can run git checkout 11f to check out this version of the application. This update also contains a database migration, so remember to run flask db upgrade after you check out the code. To ensure that you have all the dependencies installed also run pip install -r requirements/dev.txt.

Permanent Links to Blog Posts

Users may want to share links to specific blog posts with friends on social networks. For this purpose, each post will be assigned a page with a unique URL that references it. The route and view function that support permanent links are shown in Example 11-17.

Example 11-17. app/main/views.py: enabling permanent links to posts

```
@main.route('/post/<int:id>')
def post(id):
    post = Post.query.get_or_404(id)
    return render_template('post.html', posts=[post])
```

The URLs that will be assigned to blog posts are constructed with the unique `id` field assigned when the post is inserted in the database.

> For some types of applications, building permanent links that use readable URLs instead of numeric IDs may be preferred. An alternative to numeric IDs is to assign each blog post a *slug*, which is a unique string that is based on the title or first few words of the post.

Note that the *post.html* template receives a list with a single element that is the post to render. Sending a list is a matter of convenience, so that the *_posts.html* template referenced by *index.html* and *user.html* can be used in this page as well.

The permanent links are added at the bottom of each post in the generic *_posts.html* template, as shown in Example 11-18.

Example 11-18. app/templates/_posts.html: adding permanent links to posts

```
<ul class="posts">
    {% for post in posts %}
    <li class="post">
        ...
        <div class="post-content">
            ...
            <div class="post-footer">
                <a href="{{ url_for('.post', id=post.id) }}">
                    <span class="label label-default">Permalink</span>
                </a>
            </div>
        </div>
    </li>
    {% endfor %}
</ul>
```

The new *post.html* template that renders the permanent link page is shown in Example 11-19. It includes the example template.

Example 11-19. app/templates/post.html: permanent link template

```
{% extends "base.html" %}

{% block title %}Flasky - Post{% endblock %}

{% block page_content %}
{% include '_posts.html' %}
{% endblock %}
```

If you have cloned the application's Git repository on GitHub, you can run `git checkout 11g` to check out this version of the application.

Blog Post Editor

The last feature related to blog posts is a post editor that allows users to edit their own posts. The blog post editor lives in a standalone page and is also based on Flask-PageDown, so a text area where the Markdown text of the blog post can be edited is followed by a rendered preview. The *edit_post.html* template is shown in Example 11-20.

Example 11-20. app/templates/edit_post.html: edit blog post template

```
{% extends "base.html" %}
{% import "bootstrap/wtf.html" as wtf %}

{% block title %}Flasky - Edit Post{% endblock %}

{% block page_content %}
<div class="page-header">
    <h1>Edit Post</h1>
</div>
<div>
    {{ wtf.quick_form(form) }}
</div>
{% endblock %}

{% block scripts %}
{{ super() }}
{{ pagedown.include_pagedown() }}
{% endblock %}
```

The route that supports the blog post editor is shown in Example 11-21.

Example 11-21. app/main/views.py: edit blog post route

```
@main.route('/edit/<int:id>', methods=['GET', 'POST'])
@login_required
def edit(id):
    post = Post.query.get_or_404(id)
    if current_user != post.author and \
            not current_user.can(Permission.ADMIN):
        abort(403)
    form = PostForm()
    if form.validate_on_submit():
        post.body = form.body.data
        db.session.add(post)
        db.session.commit()
        flash('The post has been updated.')
        return redirect(url_for('.post', id=post.id))
    form.body.data = post.body
    return render_template('edit_post.html', form=form)
```

This view function is coded to allow only the author of a blog post to edit it, except for administrators, who are allowed to edit posts from all users. If a user tries to edit a post from another user, the view function responds with a 403 code. The `PostForm` web form class used here is the same one used on the home page.

To complete the feature, a link to the blog post editor can be added below each blog post, next to the permanent link, as shown in Example 11-22.

Example 11-22. app/templates/_posts.html: adding the edit blog post link

```
<ul class="posts">
    {% for post in posts %}
    <li class="post">
        ...
        <div class="post-content">
            ...
            <div class="post-footer">
                ...
                {% if current_user == post.author %}
                <a href="{{ url_for('.edit', id=post.id) }}">
                    <span class="label label-primary">Edit</span>
                </a>
                {% elif current_user.is_administrator() %}
                <a href="{{ url_for('.edit', id=post.id) }}">
                    <span class="label label-danger">Edit [Admin]</span>
                </a>
                {% endif %}
            </div>
        </div>
    </li>
    {% endfor %}
</ul>
```

This change adds an "Edit" link to any blog posts that are authored by the current user. For administrators, the link is added to all posts. The administrator link is styled differently as a visual cue that this is an administration feature. Figure 11-4 shows how the Edit and Permalink links look in the web browser.

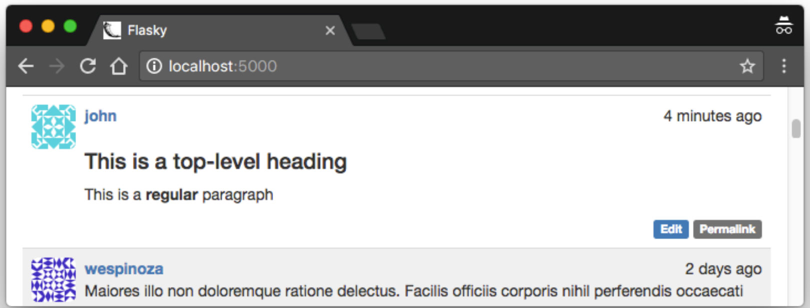


Figure 11-4. Edit and Permalink links in a blog post

If you have cloned the application's Git repository on GitHub, you can run `git checkout 11h` to check out this version of the application.

CHAPTER 12
Followers

Socially aware web applications allow users to connect with other users. Different applications call these relationships *followers*, *friends*, *contacts*, *connections*, or *buddies*, but the feature is the same regardless of the name, and in all cases involves keeping track of directional links between pairs of users and using these links in database queries.

In this chapter, you will learn how to implement a follower feature for Flasky. Users will be able to "follow" other users and choose to filter the blog post list on the home page to include only those from the users they follow.

Database Relationships Revisited

As discussed in Chapter 5, databases establish links between records using *relationships*. The one-to-many relationship is the most common type of relationship, where a record is linked with a list of related records. To implement this type of relationship, the elements on the "many" side have a foreign key that points to the linked element on the "one" side. The example application in its current state includes two one-to-many relationships: one that links user roles to lists of users and another that links users to the blog posts they authored.

Most other relationship types can be derived from the one-to-many type. The *many-to-one* relationship is a one-to-many looked at from the point of view of the "many" side. The *one-to-one* relationship type is a simplification of the one-to-many, where the "many" side is constrained to have at most one element. The only relationship type that cannot be implemented as a simple variation of the one-to-many model is the *many-to-many*, which has lists of elements on both sides. This relationship is described in detail in the following section.

Many-to-Many Relationships

The one-to-many, many-to-one, and one-to-one relationships all have at least one side with a single entity, so the links between related records are implemented with foreign keys pointing to that one element. But how do you implement a relationship where both sides are "many" sides?

Consider the classic example of a many-to-many relationship: a database of students and the classes they are taking. Clearly, you can't add a foreign key to a class in the `students` table, because a student takes many classes—one foreign key is not enough. Likewise, you cannot add a foreign key to a student in the `classes` table, because classes have more than one student. Both sides need a list of foreign keys.

The solution is to add a third table to the database, called an *association table*. Now the many-to-many relationship can be decomposed into two one-to-many relationships from each of the two original tables to the association table. Figure 12-1 shows how the many-to-many relationship between students and classes is represented.

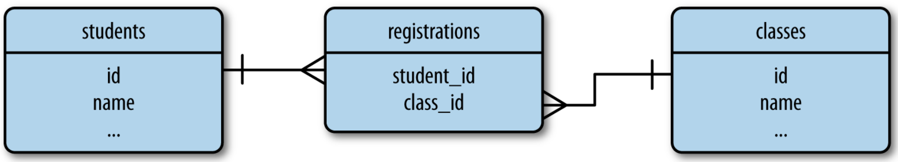


Figure 12-1. Many-to-many relationship example

The association table in this example is called `registrations`. Each row in this table represents an individual registration of a student in a class.

Querying a many-to-many relationship is a two-step process. To obtain the list of classes a student is taking, you start from the one-to-many relationship between students and registrations and get the list of registrations for the desired student. Then the one-to-many relationship between classes and registrations is traversed in the many-to-one direction to obtain all the classes associated with the registrations retrieved for the student. Likewise, to find all the students in a class, you start from the class and get a list of registrations, then get the students linked to those registrations.

Traversing two relationships to obtain query results sounds difficult, but for a simple relationship like the one in the previous example, SQLAlchemy does most of the work. Following is the code that represents the many-to-many relationship in Figure 12-1:

```
registrations = db.Table('registrations',
    db.Column('student_id', db.Integer, db.ForeignKey('students.id')),
    db.Column('class_id', db.Integer, db.ForeignKey('classes.id'))
)

class Student(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String)
    classes = db.relationship('Class',
                              secondary=registrations,
                              backref=db.backref('students', lazy='dynamic'),
                              lazy='dynamic')

class Class(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String)
```

The relationship is defined with the same `db.relationship()` construct that is used for one-to-many relationships, but in the case of a many-to-many relationship the additional `secondary` argument must be set to the association table. The relationship can be defined in either one of the two classes, with the `backref` argument taking care of exposing the relationship from the other side as well. The association table is defined as a simple table, not as a model, since SQLAlchemy manages this table internally.

The `classes` relationship uses list semantics, which makes working with a many-to-many relationship configured in this way extremely easy. Given a student `s` and a class `c`, the code that registers the student for the class is:

```
>>> s.classes.append(c)
>>> db.session.add(s)
```

The queries that list the classes student `s` is registered for and the list of students registered for class `c` are also very simple:

```
>>> s.classes.all()
>>> c.students.all()
```

The `students` relationship available in the `Class` model is the one defined in the `db.backref()` argument. Note that in this relationship the `backref` argument was expanded to also have a `lazy='dynamic'` attribute, so both sides return a query that can accept additional filters.

If student `s` later decides to drop class `c`, you can update the database as follows:

```
>>> s.classes.remove(c)
```

Self-Referential Relationships

A many-to-many relationship can be used to model users following other users, but there is a problem. In the example of students and classes, there were two very clearly defined entities linked together by the association table. However, to represent users following other users, it is just users—there is no second entity.

A relationship in which both sides belong to the same table is said to be *self-referential*. In this case the entities on the left side of the relationship are users, which can be called the "followers." The entities on the right side are also users, but these are the "followed" users. Conceptually, self-referential relationships are no different than regular relationships, but they are harder to think about. Figure 12-2 shows a database diagram for a self-referential relationship that represents users following other users.

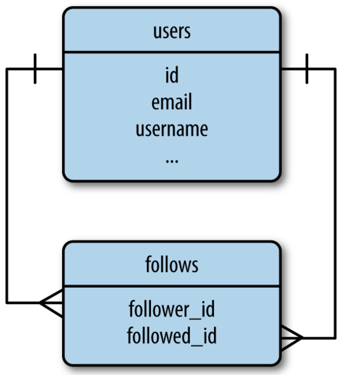


Figure 12-2. Followers, many-to-many relationship

The association table in this case is called `follows`. Each row in this table represents a user following another user. The one-to-many relationship pictured on the left side associates users with the list of "follows" rows in which they are the followers. The one-to-many relationship pictured on the right side associates users with the list of "follows" rows in which they are the followed user.

Advanced Many-to-Many Relationships

With a self-referential many-to-many relationship configured as shown in the previous example, the database can represent followers—but there is one limitation. A common need when working with many-to-many relationships is to store additional data that applies to the link between two entities. For the followers relationship, it can be useful to store the date a user started following another user, as that will enable lists of followers to be presented in chronological order. The only place this informa-

tion can be stored is in the association table, but in an implementation similar to that of the students and classes shown earlier, the association table is an internal table that is fully managed by SQLAlchemy.

To be able to work with custom data in the relationship, the association table must be promoted to a proper model that the application can access. Example 12-1 shows the new association table, represented by the `Follow` model.

Example 12-1. app/models.py: the follows association table as a model

```
class Follow(db.Model):
    __tablename__ = 'follows'
    follower_id = db.Column(db.Integer, db.ForeignKey('users.id'),
                            primary_key=True)
    followed_id = db.Column(db.Integer, db.ForeignKey('users.id'),
                            primary_key=True)
    timestamp = db.Column(db.DateTime, default=datetime.utcnow)
```

SQLAlchemy cannot use the association table transparently because that will not give the application access to the custom fields in it. Instead, the many-to-many relationship must be decomposed into the two basic one-to-many relationships for the left and right sides, and these must be defined as standard relationships. This is shown in Example 12-2.

Example 12-2. app/models.py: a many-to-many relationship implemented as two one-to-many relationships

```
class User(UserMixin, db.Model):
    # ...
    followed = db.relationship('Follow',
                               foreign_keys=[Follow.follower_id],
                               backref=db.backref('follower', lazy='joined'),
                               lazy='dynamic',
                               cascade='all, delete-orphan')
    followers = db.relationship('Follow',
                                foreign_keys=[Follow.followed_id],
                                backref=db.backref('followed', lazy='joined'),
                                lazy='dynamic',
                                cascade='all, delete-orphan')
```

Here the `followed` and `followers` relationships are defined as individual one-to-many relationships. Note that it is necessary to eliminate any ambiguity between foreign keys by specifying in each relationship which foreign key to use through the `foreign_keys` optional argument. The `db.backref()` arguments in these relationships do not apply to each other; the back references are applied to the `Follow` model.

The `lazy` argument for the back references is specified as `joined`. This lazy mode causes the related object to be loaded immediately from the join query. For example, if a user is following 100 other users, calling `user.followed.all()` will return a list of 100 `Follow` instances, where each one has the `follower` and `followed` back reference properties set to the respective users. The `lazy='joined'` mode enables this all to happen from a single database query. If `lazy` is set to the default value of `select`, then the `follower` and `followed` users are loaded lazily when they are first accessed and each attribute will require an individual query, which means that obtaining the complete list of followed users would require 100 additional database queries.

The `lazy` argument on the `User` side of both relationships has different needs. These are on the "one" side and return the "many" side; here a mode of `dynamic` is used, so that the relationship attributes return query objects instead of returning the items directly. This allows additional filters to be added to the query before it is executed.

The `cascade` argument configures how actions performed on a parent object propagate to related objects. An example of a cascade option is the rule that says that when an object is added to the database session, any objects associated with it through relationships should automatically be added to the session as well. The default cascade options are appropriate for most situations, but there is one case in which the default cascade options do not work well for this many-to-many relationship. The default cascade behavior when an object is deleted is to set the foreign key in any related objects that link to it to a null value. But for an association table, the correct behavior is to delete the entries that point to a record that was deleted, as this effectively destroys the link. This is what the `delete-orphan` cascade option does.

> The value given to `cascade` is a comma-separated list of cascade options. This is somewhat confusing, but the option named `all` represents all the cascade options except `delete-orphan`. Using the value `all, delete-orphan` leaves the default cascade options enabled and adds the delete behavior for orphans.

The application now needs to work with the two one-to-many relationships to implement the many-to-many functionality. Since these are operations that will need to be repeated often, it is a good idea to create helper methods in the `User` model for all the possible operations. The four new methods that control this relationship are shown in Example 12-3.

Example 12-3. app/models.py: followers helper methods

```
class User(db.Model):
    # ...
    def follow(self, user):
        if not self.is_following(user):
            f = Follow(follower=self, followed=user)
            db.session.add(f)

    def unfollow(self, user):
        f = self.followed.filter_by(followed_id=user.id).first()
        if f:
            db.session.delete(f)

    def is_following(self, user):
        if user.id is None:
            return False
        return self.followed.filter_by(
            followed_id=user.id).first() is not None

    def is_followed_by(self, user):
        if user.id is None:
            return False
        return self.followers.filter_by(
            follower_id=user.id).first() is not None
```

The `follow()` method manually inserts a `Follow` instance in the association table that links a follower with a followed user, giving the application the opportunity to set the custom field. The two users who are connecting are manually assigned to the new `Follow` instance in its constructor, and then the object is added to the database session as usual. Note that there is no need to manually set the `timestamp` field because it was defined with a default value that sets the current date and time. The `unfollow()` method uses the `followed` relationship to locate the `Follow` instance that links the user to the followed user who needs to be disconnected. To destroy the link between the two users, the `Follow` object is simply deleted. The `is_following()` and `is_followed_by()` methods search the left- and right-side one-to-many relationships, respectively, for the given user and return `True` if the user is found. Both ensure that the given user has been assigned an `id` before issuing a query, to avoid errors if a user that has been created but not committed to the database yet is provided.

If you have cloned the application's Git repository on GitHub, you can run `git checkout 12a` to check out this version of the application. This update contains a database migration, so remember to run `flask db upgrade` after you check out the code.

The database part of this feature is now complete. You can find a unit test that exercises the new database relationship in the source code repository on GitHub.

Followers on the Profile Page

The profile page of a user needs to present a "Follow" button if the user viewing it is not a follower, or an "Unfollow" button if the user is a follower. It is also a nice addition to show the follower and followed counts, display the lists of followers and followed users, and show a "Follows you" sign when appropriate. The changes to the user profile template are shown in Example 12-4. Figure 12-3 shows how the additions look on the profile page.

Example 12-4. app/templates/user.html: follower enhancements to the user profile header

```
{% if current_user.can(Permission.FOLLOW) and user != current_user %}
    {% if not current_user.is_following(user) %}
    <a href="{{ url_for('.follow', username=user.username) }}"
        class="btn btn-primary">Follow</a>
    {% else %}
    <a href="{{ url_for('.unfollow', username=user.username) }}"
        class="btn btn-default">Unfollow</a>
    {% endif %}
{% endif %}
<a href="{{ url_for('.followers', username=user.username) }}">
    Followers: <span class="badge">{{ user.followers.count() }}</span>
</a>
<a href="{{ url_for('.followed_by', username=user.username) }}">
    Following: <span class="badge">{{ user.followed.count() }}</span>
</a>
{% if current_user.is_authenticated and user != current_user and
    user.is_following(current_user) %}
| <span class="label label-default">Follows you</span>
{% endif %}
```

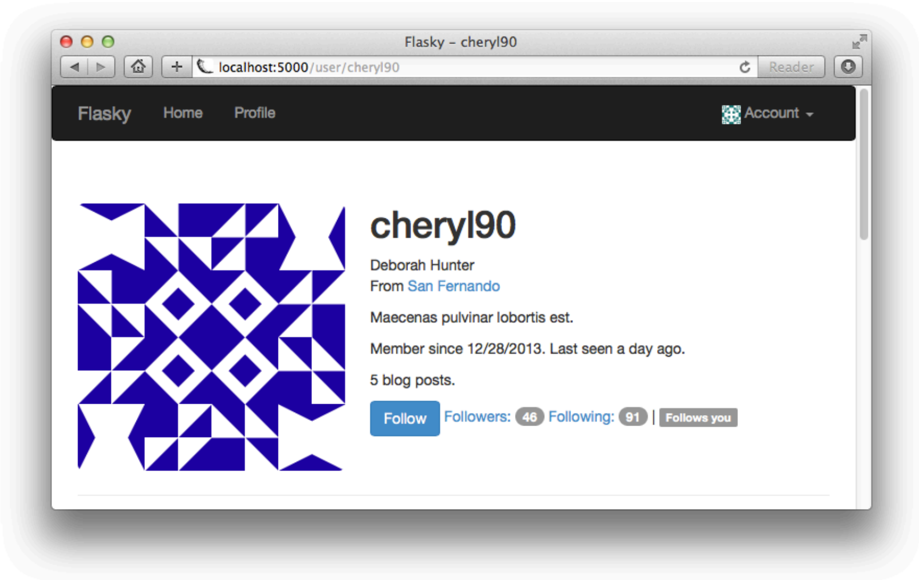


Figure 12-3. Followers on the profile page

There are four new endpoints defined in these template changes. The */follow/<username>* route is invoked when a user clicks the "Follow" button on another user's profile page. The implementation is shown in Example 12-5.

Example 12-5. app/main/views.py: follow route and view function

```
@main.route('/follow/<username>')
@login_required
@permission_required(Permission.FOLLOW)
def follow(username):
    user = User.query.filter_by(username=username).first()
    if user is None:
        flash('Invalid user.')
        return redirect(url_for('.index'))
    if current_user.is_following(user):
        flash('You are already following this user.')
        return redirect(url_for('.user', username=username))
    current_user.follow(user)
    db.session.commit()
    flash('You are now following %s.' % username)
    return redirect(url_for('.user', username=username))
```

This view function loads the requested user, verifies that it is valid and that it isn't already followed by the logged-in user, and then calls the `follow()` helper function in

the `User` model to establish the link. The */unfollow/<username>* route is implemented in a similar way.

The */followers/<username>* route is invoked when a user clicks another user's follower count on the profile page. The implementation is shown in Example 12-6.

Example 12-6. app/main/views.py: followers route and view function

```
@main.route('/followers/<username>')
def followers(username):
    user = User.query.filter_by(username=username).first()
    if user is None:
        flash('Invalid user.')
        return redirect(url_for('.index'))
    page = request.args.get('page', 1, type=int)
    pagination = user.followers.paginate(
        page, per_page=current_app.config['FLASKY_FOLLOWERS_PER_PAGE'],
        error_out=False)
    follows = [{'user': item.follower, 'timestamp': item.timestamp}
               for item in pagination.items]
    return render_template('followers.html', user=user, title="Followers of",
                           endpoint='.followers', pagination=pagination,
                           follows=follows)
```

This function loads and validates the requested user, then paginates its `followers` relationship using the same techniques learned in Chapter 11. Because the query for followers returns `Follow` instances, the list is converted into another list that has `user` and `timestamp` fields in each entry so that rendering is simpler.

The template that renders the follower list can be written generically so that it can be used for lists of followers and followed users. The template receives the user, a title for the page, the endpoint to use in the pagination links, the pagination object, and the list of results.

The `followed_by` endpoint is almost identical. The only difference is that the list of users is obtained from the `user.followed` relationship. The template arguments are also adjusted accordingly.

The *followers.html* template is implemented with a two-column table that shows usernames and their avatars on the left and Flask-Moment timestamps on the right. You can consult the source code repository on GitHub to study the implementation in detail.

If you have cloned the application's Git repository on GitHub, you can run `git checkout 12b` to check out this version of the application.

Querying Followed Posts Using a Database Join

The application's home page currently shows all the posts in the database in descending chronological order. With the followers feature now complete, it would be a nice addition to give users the option to view blog posts from only the users they follow.

The obvious way to load all the posts authored by followed users is to first get the list of those users and then get the posts from each and sort them into a single list. Of course, that approach does not scale well; the effort required to obtain this combined list will grow as the database grows, and operations such as pagination cannot be done efficiently. This problem is commonly known as the "*N*+1 problem," because working with the database in this way requires issuing *N*+1 database queries, with *N* being the number of results returned by the first query. The key to obtaining the blog posts with good performance regardless of the database size is doing it all with a single query.

The database operation that can do this is called a *join*. A join operation takes two or more tables and finds all the combinations of rows that satisfy a given condition. The resulting combined rows are inserted into a temporary table that is the result of the join. The best way to explain how joins work is through an example.

Table 12-1 shows an example `users` table with three users.

Table 12-1. users table

id	username
1	john
2	susan
3	david

Table 12-2 shows the corresponding `posts` table, with some blog posts.

Table 12-2. posts table

id	author_id	body
1	2	Blog post by susan
2	1	Blog post by john
3	3	Blog post by david
4	1	Second blog post by john

Finally, Table 12-3 shows who is following whom. In this table you can see that *john* is following *david*, *susan* is following *john* and *david*, and *david* is not following anyone.

Table 12-3. follows table

follower_id	followed_id
1	3
2	1
2	3

To obtain the list of posts by users followed by the user *susan*, the `posts` and `follows` tables must be combined. First the `follows` table is filtered to keep just the rows that have *susan* as the follower, which in this example are the last two rows. Then a temporary join table is created from all the possible combinations of rows from the `posts` and filtered `follows` tables in which the `author_id` of the post is the same as the `followed_id` of the follow, effectively selecting any posts that appear in the list of users *susan* is following. Table 12-4 shows the result of the join operation. The columns that were used to perform the join are marked with an * in this table.

Table 12-4. Joined table

id	author_id*	body	follower_id	followed_id*
2	1	Blog post by john	2	1
3	3	Blog post by david	2	3
4	1	Second blog post by john	2	1

This table contains exactly the list of blog posts authored by users that *susan* is following. The Flask-SQLAlchemy query that performs the join operation as described is fairly complex:

```
return db.session.query(Post).select_from(Follow).\
    filter_by(follower_id=self.id).\
    join(Post, Follow.followed_id == Post.author_id)
```

All the queries that you have seen so far start from the `query` attribute of the model that is queried. That format does not work well for this query, because the query needs to return `posts` rows, yet the first operation that needs to be done is to apply a filter to the `follows` table. So, a more basic form of the query is used instead. To fully understand this query, each part should be looked at individually:

- `db.session.query(Post)` specifies that this is going to be a query that returns `Post` objects.
- `select_from(Follow)` says that the query begins with the `Follow` model.
- `filter_by(follower_id=self.id)` performs the filtering of the `follows` table by the follower user.

- `join(Post, Follow.followed_id == Post.author_id)` joins the results of `filter_by()` with the `Post` objects.

The query can be simplified by swapping the order of the filter and the join:

```
return Post.query.join(Follow, Follow.followed_id == Post.author_id)\
    .filter(Follow.follower_id == self.id)
```

Issuing the join operation first means the query can be started from `Post.query`, so now the only two filters that need to be applied are `join()` and `filter()`. It may seem that doing the join first and then the filtering would be more work, but in reality these two queries are equivalent. SQLAlchemy first collects all the filters and then generates the query in the most efficient way. The native SQL instructions for these two queries are nearly identical, something that you can confirm by printing the query object converted to a string (i.e., `print(str(query))`). The final version of this query is added to the `Post` model, as shown in Example 12-7.

Example 12-7. app/models.py: obtaining followed posts

```
class User(db.Model):
    # ...
    @property
    def followed_posts(self):
        return Post.query.join(Follow, Follow.followed_id == Post.author_id)\
            .filter(Follow.follower_id == self.id)
```

Note that the `followed_posts()` method is defined as a property so that it does not need the `()`. That way, all relationships have a consistent syntax.

> If you have cloned the application's Git repository on GitHub, you can run `git checkout 12c` to check out this version of the application.

Joins are extremely hard to wrap your head around; you may need to experiment with the example code in a shell before it all sinks in.

Showing Followed Posts on the Home Page

The home page can now give users the choice to view all blog posts or just those from followed users. Example 12-8 shows how this choice is implemented.

Example 12-8. app/main/views.py: showing all or followed posts

```
@main.route('/', methods = ['GET', 'POST'])
def index():
    # ...
    show_followed = False
    if current_user.is_authenticated:
        show_followed = bool(request.cookies.get('show_followed', ''))
    if show_followed:
        query = current_user.followed_posts
    else:
        query = Post.query
    pagination = query.order_by(Post.timestamp.desc()).paginate(
        page, per_page=current_app.config['FLASKY_POSTS_PER_PAGE'],
        error_out=False)
    posts = pagination.items
    return render_template('index.html', form=form, posts=posts,
                           show_followed=show_followed, pagination=pagination)
```

The choice of showing all or followed posts is stored in a cookie called `show_followed` that, when set to a nonempty string, indicates that only followed posts should be shown. Cookies are stored in the request object as a `request.cookies` dictionary. The string value of the cookie is converted to a Boolean, and based on its value a `query` local variable is set to the query that obtains the complete or filtered list of blog posts. To show all the posts, the top-level query `Post.query` is used, and the recently added `User.followed_posts` property is used when the list should be restricted to followed users. The query stored in the `query` local variable is then paginated and the results sent to the template as before.

The `show_followed` cookie is set in two new routes, shown in Example 12-9.

Example 12-9. app/main/views.py: selection of all or followed posts

```
@main.route('/all')
@login_required
def show_all():
    resp = make_response(redirect(url_for('.index')))
    resp.set_cookie('show_followed', '', max_age=30*24*60*60)  # 30 days
    return resp

@main.route('/followed')
@login_required
def show_followed():
    resp = make_response(redirect(url_for('.index')))
    resp.set_cookie('show_followed', '1', max_age=30*24*60*60)  # 30 days
    return resp
```

Links to these routes are added to the home page template. When they are invoked, the `show_followed` cookie is set to the proper value and a redirect back to the home page is issued.

Cookies can be set only on a response object, so these routes need to create a response object through `make_response()` instead of letting Flask do this.

The `set_cookie()` function takes the cookie name and the value as the first two arguments. The `max_age` optional argument sets the number of seconds until the cookie expires. Not including this argument makes the cookie expire when the browser window is closed. In this case, a maximum age of 30 days is set so that the setting is remembered even if the user does not return to the application for several days.

The changes to the template add two navigation tabs at the top of the page that invoke the */all* or */followed* routes to set the correct settings in the session. You can inspect the template changes in detail in the source code repository on GitHub. Figure 12-4 shows how the home page looks with these changes.

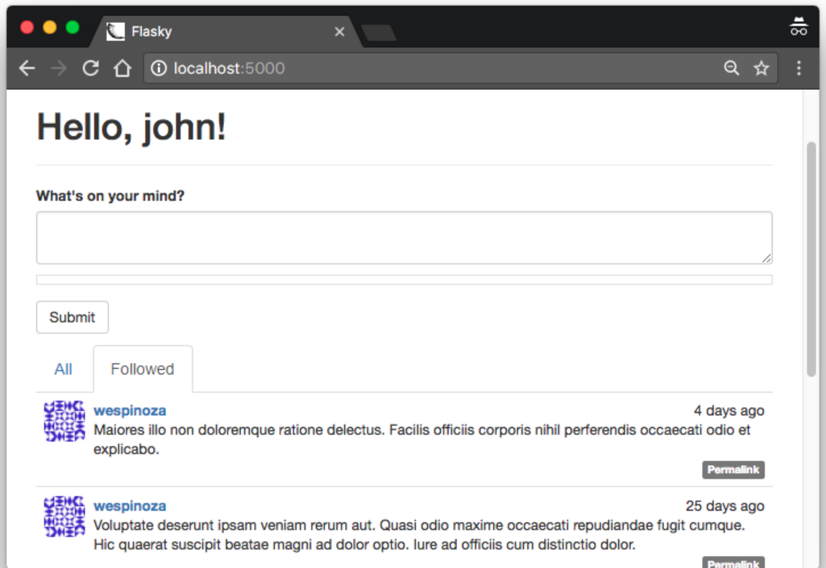


Figure 12-4. Followed posts on the home page

If you have cloned the application's Git repository on GitHub, you can run `git checkout 12d` to check out this version of the application.

If you try the application at this point and switch to the followed list of posts, you will notice that your own posts do not appear in the list. This is of course correct, because users are not followers of themselves.

Even though the queries are working as designed, most users will expect to see their own posts when they are looking at those of their friends. The easiest way to address this issue is to register all users as their own followers at the time they are created. This trick is shown in Example 12-10.

Example 12-10. app/models.py: making users their own followers when they are created

```
class User(UserMixin, db.Model):
    # ...
    def __init__(self, **kwargs):
        # ...
        self.follow(self)
```

Unfortunately, you likely have several users in the database who are already created and are not following themselves. If the database is small and easy to regenerate, then it can be deleted and re-created, but if that is not an option, then adding an update function that fixes existing users is the proper solution. This is shown in Example 12-11.

Example 12-11. app/models.py: making users their own followers

```
class User(UserMixin, db.Model):
    # ...
    @staticmethod
    def add_self_follows():
        for user in User.query.all():
            if not user.is_following(user):
                user.follow(user)
                db.session.add(user)
                db.session.commit()
    # ...
```

Now the database can be updated by running the previous example function from the shell:

```
(venv) $ flask shell
>>> User.add_self_follows()
```

Creating functions that introduce updates to the database is a common technique used to update applications that are deployed, as running a scripted update is less error prone than updating databases manually. In Chapter 17 you will see how this function and others like it can be incorporated into a deployment script.

Making all users self-followers makes the application more usable, but this change introduces a few complications. The follower and followed user counts shown in the user profile page are now increased by one due to the self-follower links. The numbers need to be decreased by one to be accurate, which is easy to do directly in the template by rendering `{{ user.followers.count() - 1 }}` and `{{ user.followed.count() - 1 }}`. The lists of followers and followed users also must be adjusted to not show the same user, another simple task to do in the template with a conditional. Finally, any unit tests that check follower counts are also affected by the self-follower links and must be adjusted to account for the self-followers.

> If you have cloned the application's Git repository on GitHub, you can run `git checkout 12e` to check out this version of the application.

In the next chapter, the user comment subsystem will be implemented—another very important feature of socially aware applications.

CHAPTER 13
User Comments

Allowing users to interact is key to the success of a social blogging platform. In this chapter, you will learn how to implement user comments. The techniques presented are generic enough to be directly applicable to a large number of socially enabled applications.

Database Representation of Comments

Comments are not very different from blog posts. Both have a body, an author, and a timestamp, and in this particular implementation both are written with Markdown syntax. Figure 13-1 shows a diagram of the `comments` table and its relationships with other tables in the database.

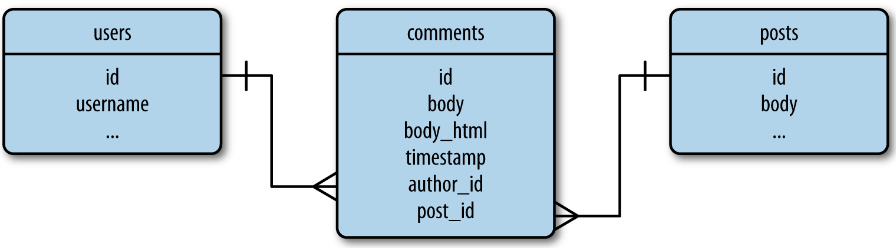


Figure 13-1. Database representation of blog post comments

Comments apply to specific blog posts, so a one-to-many relationship from the `posts` table is defined. This relationship can be used to obtain the list of comments associated with a particular blog post.

The `comments` table is also in a one-to-many relationship with the `users` table. This relationship gives access to all the comments made by a user, and indirectly how

many comments a user has written, a piece of information that can be interesting to show in user profile pages. The definition of the `Comment` model is shown in Example 13-1.

Example 13-1. app/models.py: comment model

```
class Comment(db.Model):
    __tablename__ = 'comments'
    id = db.Column(db.Integer, primary_key=True)
    body = db.Column(db.Text)
    body_html = db.Column(db.Text)
    timestamp = db.Column(db.DateTime, index=True, default=datetime.utcnow)
    disabled = db.Column(db.Boolean)
    author_id = db.Column(db.Integer, db.ForeignKey('users.id'))
    post_id = db.Column(db.Integer, db.ForeignKey('posts.id'))

    @staticmethod
    def on_changed_body(target, value, oldvalue, initiator):
        allowed_tags = ['a', 'abbr', 'acronym', 'b', 'code', 'em', 'i',
                        'strong']
        target.body_html = bleach.linkify(bleach.clean(
            markdown(value, output_format='html'),
            tags=allowed_tags, strip=True))

db.event.listen(Comment.body, 'set', Comment.on_changed_body)
```

The attributes of the `Comment` model are almost the same as those of `Post`. One addition is the `disabled` field, a Boolean that will be used by moderators to suppress comments that are inappropriate or offensive. Like blog posts, comments define an event that triggers any time the `body` field changes, automating the rendering of the Markdown text to HTML. The process is identical to what was done for blog posts in Chapter 11, but since comments tend to be short, the list of HTML tags that are allowed in the conversion from Markdown is more restrictive, the paragraph-related tags have been removed, and only the character formatting tags are left.

To complete the database changes, the `User` and `Post` models must define the one-to-many relationships with the `comments` table, as shown in Example 13-2.

Example 13-2. app/models.py: one-to-many relationships from users and posts to comments

```
class User(db.Model):
    # ...
    comments = db.relationship('Comment', backref='author', lazy='dynamic')

class Post(db.Model):
    # ...
    comments = db.relationship('Comment', backref='post', lazy='dynamic')
```

Comment Submission and Display

In this application, comments are displayed on the individual blog post pages that were added as permanent links in Chapter 11. A submission form is also included on these pages. Example 13-3 shows the web form that will be used to enter comments—an extremely simple form that only has a text field and a submit button.

Example 13-3. app/main/forms.py: comment input form

```
class CommentForm(FlaskForm):
    body = StringField('', validators=[DataRequired()])
    submit = SubmitField('Submit')
```

Example 13-4 shows the updated */post/<int:id>* route with support for comments.

Example 13-4. app/main/views.py: blog post comments support

```
@main.route('/post/<int:id>', methods=['GET', 'POST'])
def post(id):
    post = Post.query.get_or_404(id)
    form = CommentForm()
    if form.validate_on_submit():
        comment = Comment(body=form.body.data,
                          post=post,
                          author=current_user._get_current_object())
        db.session.add(comment)
        db.session.commit()
        flash('Your comment has been published.')
        return redirect(url_for('.post', id=post.id, page=-1))
    page = request.args.get('page', 1, type=int)
    if page == -1:
        page = (post.comments.count() - 1) // \
               current_app.config['FLASKY_COMMENTS_PER_PAGE'] + 1
    pagination = post.comments.order_by(Comment.timestamp.asc()).paginate(
        page, per_page=current_app.config['FLASKY_COMMENTS_PER_PAGE'],
        error_out=False)
    comments = pagination.items
    return render_template('post.html', posts=[post], form=form,
                           comments=comments, pagination=pagination)
```

This view function instantiates the comment form and sends it to the *post.html* template for rendering. The logic that inserts a new comment when the form is submitted is similar to the handling of blog posts. As in the `Post` case, the `author` of the comment cannot be set directly to `current_user` because this is a context variable proxy object. The expression `current_user._get_current_object()` returns the actual `User` object.

The comments are sorted by their timestamp in chronological order, so new comments are always added at the bottom of the list. When a new comment is entered, the redirect that ends the request goes back to the same URL, but the `url_for()` function sets the `page` to `-1`, a special page number that is used to request the last page of comments so that the comment just entered is seen on the page. When the page number is obtained from the query string and found to be `-1`, a calculation with the number of comments and the page size is done to obtain the actual page number to use.

The list of comments associated with the post is obtained through the `post.comments` one-to-many relationship, sorted by comment timestamp, and paginated with the same techniques used for blog posts. The comments and the pagination object are sent to the template for rendering. The `FLASKY_COMMENTS_PER_PAGE` configuration variable is added to *config.py* to control the size of each page of comments.

The comment rendering is defined in a new template, *_comments.html*, that is similar to *_posts.html* but uses a different set of CSS classes. This template is included by *_posts.html* below the body of the post, followed by a call to the pagination macro. You can review the changes to the templates in the application's GitHub repository.

To complete this feature, blog posts shown on the home and profile pages need links to the pages with the comments. This is shown in Example 13-5.

Example 13-5. _app/templates/_posts.html: link to blog post comments

```
<a href="{{ url_for('.post', id=post.id) }}#comments">
    <span class="label label-primary">
        {{ post.comments.count() }} Comments
    </span>
</a>
```

Note how the text of the link includes the number of comments, which is easily obtained from the one-to-many relationship between the `posts` and `comments` tables using SQLAlchemy's `count()` filter.

Also of interest is the structure of the link to the comments page, which is built as the permanent link for the post with a *#comments* suffix added. This last part is called a *URL fragment* and is used to indicate an initial scroll position for the page. The web browser looks for an element with the `id` given and scrolls the page so that element appears at the top of the page. This initial position is set to the "Comments" heading in the *post.html* template, which is written as `<h4 id="comments">Comments</h4>`. Figure 13-2 shows how the comments appear on the page.

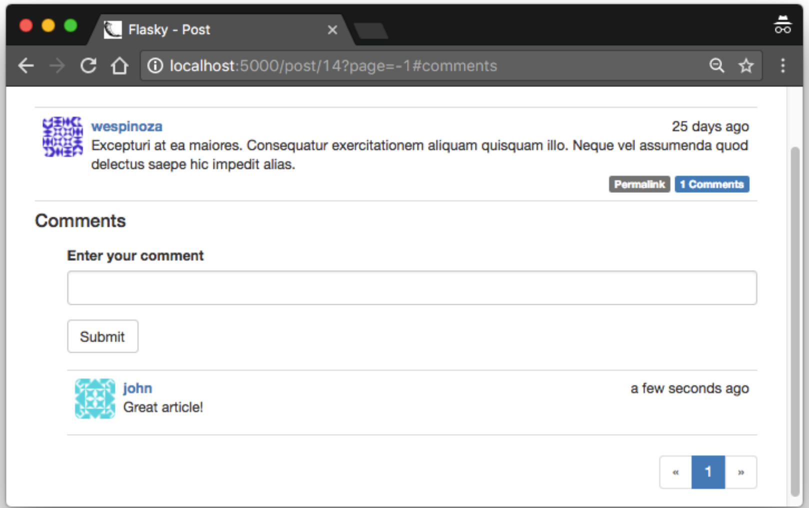


Figure 13-2. Blog post comments

An additional change was made to the pagination macro. The pagination links for comments also need the *#comments* fragment added, so a fragment argument was added to the macro and passed in the macro invocation from the *post.html* template.

> If you have cloned the application's Git repository on GitHub, you can run `git checkout 13a` to check out this version of the application. This update contains a database migration, so remember to run `flask db upgrade` after you check out the code.

Comment Moderation

In Chapter 9 a list of user roles was defined, each with a list of permissions. One of the permissions was `Permission.MODERATE`, which gives users who have it in their roles the power to moderate comments made by others.

This feature will be exposed as a link in the navigation bar that appears only to users who are permitted to use it. This is done in the *base.html* template using a conditional, as shown in Example 13-6.

Example 13-6. app/templates/base.html: Moderate Comments link in navigation bar

```
...
{% if current_user.can(Permission.MODERATE) %}
<li><a href="{{ url_for('main.moderate') }}">Moderate Comments</a></li>
{% endif %}
...
```

The moderation page shows the comments for all the posts in the same list, with the most recent comments shown first. Below each comment is a button that can toggle the `disabled` attribute. The */moderate* route is shown in Example 13-7.

Example 13-7. app/main/views.py: comment moderation route

```
@main.route('/moderate')
@login_required
@permission_required(Permission.MODERATE)
def moderate():
    page = request.args.get('page', 1, type=int)
    pagination = Comment.query.order_by(Comment.timestamp.desc()).paginate(
        page, per_page=current_app.config['FLASKY_COMMENTS_PER_PAGE'],
        error_out=False)
    comments = pagination.items
    return render_template('moderate.html', comments=comments,
                           pagination=pagination, page=page)
```

This is a very simple function that reads a page of comments from the database and passes them on to a template for rendering. Along with the comments, the template receives the pagination object and the current page number.

The *moderate.html* template, shown in Example 13-8, is also simple because it relies on the *_comments.html* subtemplate created earlier for the rendering of the comments.

Example 13-8. app/templates/moderate.html: comment moderation template

```
{% extends "base.html" %}
{% import "_macros.html" as macros %}

{% block title %}Flasky - Comment Moderation{% endblock %}

{% block page_content %}
<div class="page-header">
    <h1>Comment Moderation</h1>
</div>
{% set moderate = True %}
{% include '_comments.html' %}
{% if pagination %}
<div class="pagination">
```

```
    {{ macros.pagination_widget(pagination, '.moderate') }}
</div>
{% endif %}
{% endblock %}
```

This template defers the rendering of the comments to the *_comments.html* template, but before it hands control to the subordinate template it uses Jinja2's `set` directive to define a `moderate` template variable set to `True`. This variable is used by the *_comments.html* template to determine whether the moderation features need to be rendered.

The portion of the *_comments.html* template that renders the body of each comment needs to be modified in two ways. For regular users (when the `moderate` variable is not set), any comments that are marked as disabled should be suppressed. For moderators (when `moderate` is set to `True`), the body of the comment must be rendered regardless of the disabled state, and below the body a button should be included to toggle the state. Example 13-9 shows these changes.

Example 13-9. app/templates/_comments.html: rendering of the comment bodies

```
...
<div class="comment-body">
    {% if comment.disabled %}
    <p></p><i>This comment has been disabled by a moderator.</i></p>
    {% endif %}
    {% if moderate or not comment.disabled %}
        {% if comment.body_html %}
            {{ comment.body_html | safe }}
        {% else %}
            {{ comment.body }}
        {% endif %}
    {% endif %}
</div>
{% if moderate %}
    <br>
    {% if comment.disabled %}
    <a class="btn btn-default btn-xs" href="{{ url_for('.moderate_enable',
        id=comment.id, page=page) }}">Enable</a>
    {% else %}
    <a class="btn btn-danger btn-xs" href="{{ url_for('.moderate_disable',
        id=comment.id, page=page) }}">Disable</a>
    {% endif %}
{% endif %}
...
```

With these changes, users will see a short notice for disabled comments. Moderators will see both the notice and the comment body. Moderators will also see a button to toggle the disabled state below each comment. The button invokes one of two new

routes, depending on which of the two possible states the comment is changing to. Example 13-10 shows how these routes are defined.

Example 13-10. app/main/views.py: comment moderation routes

```
@main.route('/moderate/enable/<int:id>')
@login_required
@permission_required(Permission.MODERATE)
def moderate_enable(id):
    comment = Comment.query.get_or_404(id)
    comment.disabled = False
    db.session.add(comment)
    return redirect(url_for('.moderate',
                            page=request.args.get('page', 1, type=int)))

@main.route('/moderate/disable/<int:id>')
@login_required
@permission_required(Permission.MODERATE)
def moderate_disable(id):
    comment = Comment.query.get_or_404(id)
    comment.disabled = True
    db.session.add(comment)
    return redirect(url_for('.moderate',
                            page=request.args.get('page', 1, type=int)))
```

The comment enable and disable routes load the comment object, set the `disabled` field to the proper value, and write it back to the database. At the end, they redirect back to the comment moderation page (shown in Figure 13-3), and if a `page` argument was given in the query string, they include it in the redirect. The buttons in the *_comments.html* template are rendered with the `page` argument so that the redirect brings the user back to the same page.

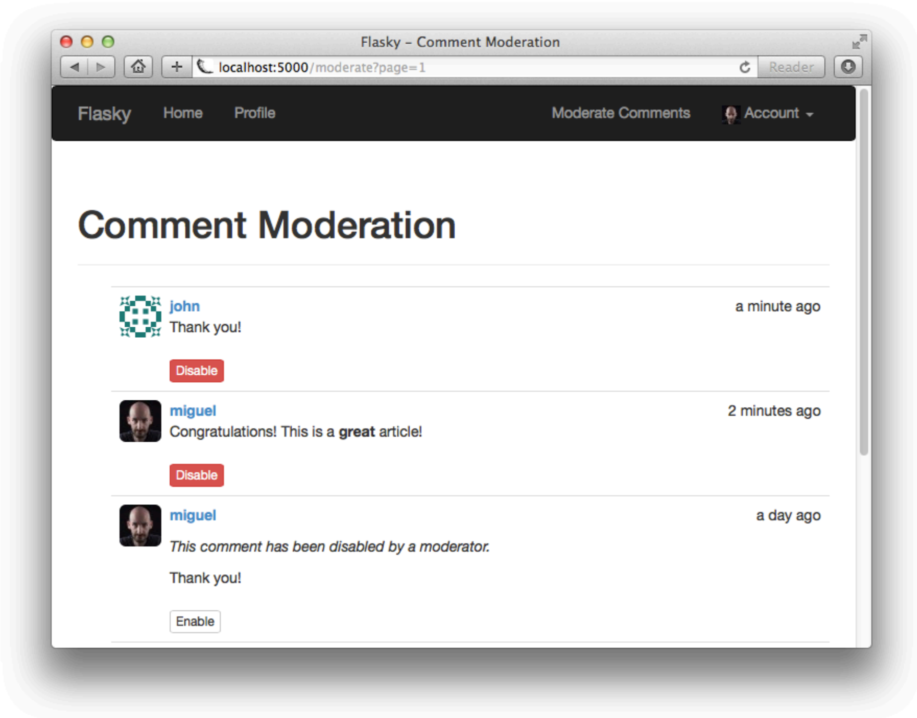


Figure 13-3. Comment moderation page

> If you have cloned the application's Git repository on GitHub, you can run `git checkout 13b` to check out this version of the application.

The topic of social features is completed with this chapter. In the next chapter, you will learn how to expose the application functionality as an API that clients such as smartphone apps can use.

CHAPTER 14

Application Programming Interfaces

In recent years, there has been a trend in web applications to move more and more of the business logic to the client side, producing an architecture that is known as Rich Internet Applications (RIAs). In RIAs, the server's main (and sometimes only) function is to provide the client application with data retrieval and storage services. In this model, the server becomes a *web service* or *application programming interface* (API).

There are several protocols by which RIAs can communicate with a web service. Remote procedure call (RPC) protocols such as XML-RPC or its derivative, the Simplified Object Access Protocol (SOAP), were popular choices a few years ago. More recently, the Representational State Transfer (REST) architecture has emerged as the favorite for web applications due to its being built on the familiar model of the World Wide Web.

Flask is an ideal framework to build RESTful web services, thanks to its lightweight nature. In this chapter, you will learn how to implement a Flask-based RESTful API.

Introduction to REST

Roy Fielding's PhD dissertation describes the REST architectural style for web services in terms of its six defining characteristics:

Client–server
: There must be a clear separation between clients and servers.

Stateless
: A client request must contain all the information that is necessary to carry it out. The server must not store any state about the client that persists from one request to the next.

Cache
: Responses from the server can be labeled as cacheable or noncacheable so that clients (or intermediaries between clients and servers) can use a cache for optimization purposes.

Uniform interface
: The protocol by which clients access server resources must be consistent, well defined, and standardized. This is the most complex aspect of REST, covering the use of unique resource identifiers, resource representations, self-descriptive messages between client and server, and hypermedia.

Layered system
: Proxy servers, caches, or gateways can be inserted between clients and servers as necessary to improve performance, reliability, and scalability.

Code-on-demand
: Clients can optionally download code from the server to execute in their context.

Resources Are Everything

The concept of *resources* is core to the REST architectural style. In this context, a resource is an item of interest in the domain of the application. For example, in the blogging application, users, blog posts, and comments are all resources.

Each resource must have a unique identifier that represents it. When working with HTTP, identifiers for resources are URLs. Continuing with the blogging example, a blog post could be represented by the URL */api/posts/12345*, where *12345* is the identifier for a post, such as the post's database primary key. The format or contents of the URL do not really matter; all that matters is that each resource URL uniquely identifies a resource.

A collection of all the resources in a class also has an assigned URL. The URL for the collection of blog posts could be */api/posts/* and the URL for the collection of all comments could be */api/comments/*.

An API can also define collection URLs that represent logical subsets of all the resources in a class. For example, the collection of all comments in blog post 12345 could be represented by the URL */api/posts/12345/comments/*. It is common to define URLs that represent collections of resources with a trailing slash, as this gives them a "subdirectory" representation.

> Be aware that Flask applies special treatment to routes that end with a slash. If a client requests a URL without a trailing slash and there is a matching route that has a slash at the end, then Flask will automatically respond with a redirect to the trailing-slash URL. No redirects are issued for the reverse case.

Request Methods

The client application sends requests to the server at the established resource URLs and uses the request *method* to indicate the desired operation. To obtain the list of available blog posts in the blogging API the client would send a `GET` request to *http://www.example.com/api/posts/*, and to insert a new blog post it would send a `POST` request to the same URL, with the contents of the blog post in the request body. To retrieve blog post 12345 the client would send a `GET` request to *http://www.example.com/api/posts/12345*. Table 14-1 lists the request methods that are commonly used in RESTful APIs, with their meanings.

Table 14-1. HTTP request methods in RESTful APIs

Request method	Target	Description	HTTP response status code
`GET`	Individual resource URL	Obtain the resource.	200
`GET`	Resource collection URL	Obtain the collection of resources (or one page from it if the server implements pagination).	200
`POST`	Resource collection URL	Create a new resource and add it to the collection. The server chooses the URL of the new resource and returns it in a `Location` header in the response.	201
`PUT`	Individual resource URL	Modify an existing resource. Alternatively, this method can also be used to create a new resource when the client can choose the resource URL.	200 or 204
`DELETE`	Individual resource URL	Delete a resource.	200 or 204
`DELETE`	Resource collection URL	Delete all resources in the collection.	200 or 204

The REST architecture does not require that all methods be implemented for a resource. If the client invokes a method that is not supported for a given resource, then a response with the 405 status code (Method Not Allowed) should be returned. Flask handles this error automatically.

The `GET`, `POST`, `PUT`, and `DELETE` request methods are not the only ones. The HTTP protocol relies on other methods, such as `HEAD` and `OPTIONS`, which are automatically implemented by Flask.

Request and Response Bodies

Resources are sent back and forth between client and server in the bodies of requests and responses, but REST does not specify the format to use to encode resources. The `Content-Type` header in requests and responses is used to indicate the format in which a resource is encoded in the body. The standard content negotiation mecha-

nisms in the HTTP protocol can be used between client and server to agree on a format that both support.

The two formats commonly used with RESTful web services are JavaScript Object Notation (JSON) and Extensible Markup Language (XML). For web-based RIAs, JSON is attractive due to being much more concise than XML, and because of its close ties to JavaScript, the client-side scripting language used by web browsers. Returning to the blog example API, a blog post resource could have the following JSON representation:

```
{
    "self_url": "http://www.example.com/api/posts/12345",
    "title": "Writing RESTful APIs in Python",
    "author_url": "http://www.example.com/api/users/2",
    "body": "... text of the article here ...",
    "comments_url": "http://www.example.com/api/posts/12345/comments"
}
```

Note how the `url`, `author_url`, and `comments_url` fields are fully qualified resource URLs. This is important because these URLs allow the client to discover new resources.

In a well-designed RESTful API, the client knows a short list of top-level resource URLs and then discovers the rest from links included in responses, similar to how you can discover new web pages while browsing the web by clicking on links that appear in pages that you know about.

Versioning

In a traditional server-centric web application, the server has full control of the application. When an application is updated, installing the new version on the server is enough to update all users because even the parts of the application that run in the user's web browser are downloaded from the server.

The situation with RIAs and web services is more complicated, because often clients are developed independently of the server—maybe even by different people. Consider the case of an application where the RESTful web service is used by a variety of clients including web browsers and native smartphone clients. The web browser client can be updated on the server at any time, but the smartphone apps cannot be updated by force; the smartphone owner needs to allow the update to happen. Even if the smartphone owner is willing to update, it is not possible to orchestrate the upgrade of all existing instances of smartphone applications to coincide exactly with the deployment of the new server version.

For these reasons, web services need to be more tolerant than regular web applications and be able to work with old versions of their clients. Changes to a web service must be done with extreme care, because backward-incompatible changes can cause

existing clients to break until they are upgraded. A common practice is to give web services a *version*, which is added to all URLs defined in that version of the server application. For example, the first release of the blogging web service could expose the collection of blog posts at */api/v1/posts/*.

Including the web service version in the URL helps keep old and new features organized so that the server can provide new features to new clients while continuing to support old clients. An update to the blogging service could change the JSON format of blog posts and now expose blog posts as */api/v2/posts/*, while keeping the older JSON format for clients that connect to */api/v1/posts/*.

Although supporting multiple versions of the server can become a maintenance burden, there are situations in which this is the only way to allow the application to grow without causing problems to existing deployments. Older service versions can be deprecated and later removed, once all clients have migrated to a newer version.

RESTful Web Services with Flask

Flask makes it very easy to create RESTful web services. The familiar `route()` decorator along with its `methods` optional argument can be used to declare the routes that handle the resource URLs exposed by the service. Working with JSON data is also simple, as JSON data included with a request can be obtained in dictionary format by calling `request.get_json()`, and a response that needs to contain JSON can be easily generated from a Python dictionary using Flask's `jsonify()` helper function.

The following sections show how Flasky can be extended with a RESTful web service that gives clients access to blog posts and related resources.

Creating an API Blueprint

The routes associated with a RESTful API form a self-contained subset of the application, so putting them in their own blueprint is the best way to keep them well organized. The general structure of the API blueprint within the application is shown in Example 14-1.

Example 14-1. API blueprint structure

```
|-flasky
  |-app/
    |-api
      |-__init__.py
      |-users.py
      |-posts.py
      |-comments.py
      |-authentication.py
      |-errors.py
      |-decorators.py
```

Note how the package used for the API includes a version number in its name. If in the future a backward-incompatible version of the API needs to be introduced, it can be added as a separate package with a different version number and both APIs can be included in the application.

The API blueprint implements each resource in a separate module. Modules to take care of authentication and error handling and to provide custom decorators are also included. The blueprint constructor is shown in Example 14-2.

Example 14-2. app/api/__init__.py: API blueprint creation

```
from flask import Blueprint

api = Blueprint('api', __name__)

from . import authentication, posts, users, comments, errors
```

The structure of the blueprint package constructor is similar to that of the other blueprints. Importing all the components of the blueprint is necessary so that routes and other handlers are registered. Since many of these modules need to import the `api` blueprint referenced here, the imports are done at the bottom to help prevent errors due to circular dependencies.

The registration of the API blueprint is shown in Example 14-3.

Example 14-3. app/init.py: API blueprint registration

```
def create_app(config_name):
    # ...
    from .api import api as api_blueprint
    app.register_blueprint(api_blueprint, url_prefix='/api/v1')
    # ...
```

The API blueprint is registered with a URL prefix, so that all its routes will have their URLs prefixed with */api/v1*. Adding a prefix when registering the blueprint is a good idea because it eliminates the need to hardcode the version number in every blueprint route.

Error Handling

A RESTful web service informs the client of the status of a request by sending the appropriate HTTP status code in the response, plus any additional information in the response body. The typical status codes that a client can expect to see from a web service are listed in Table 14-2.

Table 14-2. HTTP response status codes typically returned by APIs

HTTP status code	Name	Description
200	OK	The request was completed successfully.
201	Created	The request was completed successfully and a new resource was created as a result.
202	Accepted	The request was accepted for processing, but it is still in progress and will run asynchronously.
204	No Content	The request was completed successfully and there is no data to return in the response.
400	Bad Request	The request is invalid or inconsistent.
401	Unauthorized	The request does not include authentication information or the credentials provided are invalid.
403	Forbidden	The authentication credentials sent with the request are insufficient for the request.
404	Not Found	The resource referenced in the URL was not found.
405	Method Not Allowed	The method requested is not supported for the given resource.
500	Internal Server Error	An unexpected error occurred while processing the request.

The handling of status codes 404 and 500 presents a small complication, in that these errors are normally generated by Flask on its own, and will return an HTML response. This can confuse an API client, which will likely expect all responses in JSON format.

One way to generate appropriate responses for all clients is to make the error handlers adapt their responses based on the format requested by the client, a technique called *content negotiation*. Example 14-4 shows an improved 404 error handler that responds with JSON to web service clients and with HTML to others. The 500 error handler is written in a similar way.

Example 14-4. app/api/errors.py: 404 error handler with HTTP content negotiation

```
@main.app_errorhandler(404)
def page_not_found(e):
    if request.accept_mimetypes.accept_json and \
            not request.accept_mimetypes.accept_html:
        response = jsonify({'error': 'not found'})
        response.status_code = 404
        return response
    return render_template('404.html'), 404
```

This new version of the error handler checks the `Accept` request header, which is decoded into `request.accept_mimetypes`, to determine what format the client wants the response in. Browsers generally do not specify any restrictions on response formats, but API clients typically do. The JSON response is generated only for clients that include JSON in their list of accepted formats, but not HTML.

The remaining status codes are generated explicitly by the web service, so they can be implemented as helper functions inside the blueprint in the *errors.py* module. Example 14-5 shows the implementation of the 403 error; the others are similar.

Example 14-5. app/api/errors.py: API error handler for status code 403

```
def forbidden(message):
    response = jsonify({'error': 'forbidden', 'message': message})
    response.status_code = 403
    return response
```

View functions in the API blueprint can invoke these auxiliary functions to generate error responses when necessary.

User Authentication with Flask-HTTPAuth

Web services, like regular web applications, need to protect information and ensure that it is not given to unauthorized parties. For this reason, RIAs must ask their users for login credentials and pass them to the server for verification.

It was mentioned earlier that one of the characteristics of RESTful web services is that they are *stateless*, which means that the server is not allowed to "remember" anything about the client between requests. Clients need to provide all the information necessary to carry out a request in the request itself, so all requests must include user credentials.

The current login functionality implemented with the help of Flask-Login stores data in the user session, which Flask stores by default in a client-side cookie, so the server does not store any user-related information; it asks the client to store it instead. It would appear that this implementation complies with the stateless requirement of REST, but the use of cookies in RESTful web services falls into a gray area, as it can be cumbersome for clients that are not web browsers to implement them. For that reason, it is generally seen as a bad design choice to use cookies in APIs.

> The stateless requirement of REST may seem overly strict, but it is not arbitrary. Stateless servers can *scale* very easily. If servers store information about clients, it is necessary to ensure that the same server always gets requests from a given client, or else to use shared storage for client data. Both are complex problems to solve that do not exist when the server is stateless.

Because the RESTful architecture is based on the HTTP protocol, *HTTP authentication* is the preferred method used to send credentials, either in its Basic or Digest flavor. With HTTP authentication, user credentials are included in an `Authorization` header with all requests.

The HTTP authentication protocol is simple enough that it can be implemented directly, but the Flask-HTTPAuth extension provides a convenient wrapper that hides the protocol details in a decorator similar to Flask-Login's `login_required`.

Flask-HTTPAuth is installed with *pip*:

```
(venv) $ pip install flask-httpauth
```

To initialize the extension for HTTP Basic authentication, an object of class `HTTPBasicAuth` must be created. Like Flask-Login, Flask-HTTPAuth makes no assumptions about the procedure required to verify user credentials, so this information is given in a callback function. Example 14-6 shows how the extension is initialized and provided with a verification callback.

Example 14-6. app/api/authentication.py: Flask-HTTPAuth initialization

```python
from flask_httpauth import HTTPBasicAuth
auth = HTTPBasicAuth()

@auth.verify_password
def verify_password(email, password):
    if email == '':
        return False
    user = User.query.filter_by(email = email).first()
    if not user:
        return False
    g.current_user = user
    return user.verify_password(password)
```

Because this type of user authentication will be used only in the API blueprint, the Flask-HTTPAuth extension is initialized in the blueprint package, and not in the application package like other extensions.

The email and password are verified using the existing support in the `User` model. The verification callback returns `True` when the login is valid and `False` otherwise. The Flask-HTTPAuth extension also will invoke the callback for requests that carry no authentication, setting both arguments to the empty string. In this case, when `email` is an empty string, the function immediately returns `False` to block the request; for certain applications it may be acceptable to allow the anonymous user by returning `True`. The authentication callback saves the authenticated user in Flask's `g` context variable so that the view function can access it later.

> Because user credentials are being exchanged with every request, it is extremely important that the API routes are exposed over secure HTTP so that all requests and responses are encrypted in transit.

When the authentication credentials are invalid, the server returns a 401 status code response to the client. Flask-HTTPAuth generates a response with this status code by default, but to ensure that the response is consistent with other errors returned by the API, the error response can be customized as shown in Example 14-7.

Example 14-7. _app/api/authentication.py: Flask-HTTPAuth error handler

```
from .errors import unauthorized

@auth.error_handler
def auth_error():
    return unauthorized('Invalid credentials')
```

To protect a route, the `auth.login_required` decorator is used:

```
    @api.route('/posts/')
    @auth.login_required
    def get_posts():
        pass
```

But since all the routes in the blueprint need to be protected in the same way, the `login_required` decorator can be included once in a `before_request` handler for the blueprint, as shown in Example 14-8.

Example 14-8. app/api/authentication.py: before_request handler with authentication

```
from .errors import forbidden

@api.before_request
@auth.login_required
def before_request():
    if not g.current_user.is_anonymous and \
            not g.current_user.confirmed:
        return forbidden('Unconfirmed account')
```

Now the authentication checks will be done automatically for all the routes in the blueprint. As an additional check, the `before_request` handler also rejects authenticated users who have not confirmed their accounts.

Token-Based Authentication

Clients must send authentication credentials with every request. To avoid having to constantly transfer sensitive information such as a password, a token-based authentication solution can be used.

In token-based authentication, the client requests an access token by sending a request that includes the login credentials as authentication. The token can then be used in place of the login credentials to authenticate requests. For security reasons,

tokens are issued with an associated expiration. When a token expires, the client must reauthenticate to get a new one. The risk of a token getting into the wrong hands is limited due to its short lifespan. Example 14-9 shows the two new methods added to the `User` model that support generation and verification of authentication tokens using `itsdangerous`.

Example 14-9. app/models.py: token-based authentication support

```
class User(db.Model):
    # ...
    def generate_auth_token(self, expiration):
        s = Serializer(current_app.config['SECRET_KEY'],
                       expires_in=expiration)
        return s.dumps({'id': self.id}).decode('utf-8')

    @staticmethod
    def verify_auth_token(token):
        s = Serializer(current_app.config['SECRET_KEY'])
        try:
            data = s.loads(token)
        except:
            return None
        return User.query.get(data['id'])
```

The `generate_auth_token()` method returns a signed token that encodes the user's `id` field. An expiration time given in seconds is also used. The `verify_auth_token()` method takes a token and, if it's found to be valid, returns the user stored in it. This is a static method, as the user will be known only after the token is decoded.

To authenticate requests that come with a token, the `verify_password` callback for Flask-HTTPAuth must be modified to accept tokens as well as regular credentials. The updated callback is shown in Example 14-10.

Example 14-10. app/api/authentication.py: improved authentication verification with token support

```
@auth.verify_password
def verify_password(email_or_token, password):
    if email_or_token == '':
        return False
    if password == '':
        g.current_user = User.verify_auth_token(email_or_token)
        g.token_used = True
        return g.current_user is not None
    user = User.query.filter_by(email=email_or_token).first()
    if not user:
        return False
    g.current_user = user
```

```
    g.token_used = False
    return user.verify_password(password)
```

In this new version, the first authentication argument can be the email address or an authentication token. If this field is blank, an anonymous user is assumed, as before. If the password is blank, then the `email_or_token` field is assumed to be a token and validated as such. If both fields are nonempty then regular email and password authentication is assumed. With this implementation, token-based authentication is optional; it is up to each client to use it or not. To give view functions the ability to distinguish between the two authentication methods a `g.token_used` variable is added.

The route that returns authentication tokens to the client is also added to the API blueprint. The implementation is shown in Example 14-11.

Example 14-11. app/api/authentication.py: authentication token generation

```
@api.route('/tokens/', methods=['POST'])
def get_token():
    if g.current_user.is_anonymous or g.token_used:
        return unauthorized('Invalid credentials')
    return jsonify({'token': g.current_user.generate_auth_token(
        expiration=3600), 'expiration': 3600})
```

Since this route is in the blueprint, the authentication mechanisms added to the `before_request` handler also apply to it. To prevent clients from authenticating to this route using a previously obtained token instead of an email address and password, the `g.token_used` variable is checked, and requests authenticated with a token are rejected. The purpose of this is to prevent users from bypassing the token expiration by requesting a new token using the old token as authentication. The function returns a token in the JSON response with a validity period of one hour. The period is also included in the JSON response.

Serializing Resources to and from JSON

A frequent need when writing a web service is to convert internal representations of resources to and from JSON, which is the transport format used in HTTP requests and responses. The process of converting an internal representation to a transport format such as JSON is called *serialization*. Example 14-12 shows a new `to_json()` method added to the `Post` class.

Example 14-12. app/models.py: converting a post to a JSON serializable dictionary

```
class Post(db.Model):
    # ...
    def to_json(self):
        json_post = {
            'url': url_for('api.get_post', id=self.id),
            'body': self.body,
            'body_html': self.body_html,
            'timestamp': self.timestamp,
            'author_url': url_for('api.get_user', id=self.author_id),
            'comments_url': url_for('api.get_post_comments', id=self.id),
            'comment_count': self.comments.count()
        }
        return json_post
```

The `url`, `author_url`, and `comments_url` fields need to return the URLs for the respective resources, so these are generated with `url_for()` calls to other routes that will be defined in the API blueprint.

This example shows how it is possible to return "made-up" attributes in the representation of a resource. The `comment_count` field returns the number of comments that exist for the blog post. Although this is not a real attribute of the model, it is included in the resource representation as a convenience to the client.

The `to_json()` method for `User` models can be constructed in a similar way. This method is shown in Example 14-13.

Example 14-13. app/models.py: converting a user to a JSON serializable dictionary

```
class User(UserMixin, db.Model):
    # ...
    def to_json(self):
        json_user = {
            'url': url_for('api.get_user', id=self.id),
            'username': self.username,
            'member_since': self.member_since,
            'last_seen': self.last_seen,
            'posts_url': url_for('api.get_user_posts', id=self.id),
            'followed_posts_url': url_for('api.get_user_followed_posts',
                                          id=self.id),
            'post_count': self.posts.count()
        }
        return json_user
```

Note how in this method some of the attributes of the user, such as `email` and `role`, are omitted from the response for privacy reasons. This example again demonstrates

that the representation of a resource offered to clients does not need to be identical to the internal definition of the corresponding database model.

The inverse of serialization is called *deserialization*. Deserializing a JSON structure back to a model presents the challenge that some of the data coming from the client might be invalid, wrong, or unnecessary. Example 14-14 shows the method that creates a `Post` from JSON.

Example 14-14. app/models.py: creating a blog post from JSON

```
from app.exceptions import ValidationError

class Post(db.Model):
    # ...
    @staticmethod
    def from_json(json_post):
        body = json_post.get('body')
        if body is None or body == '':
            raise ValidationError('post does not have a body')
        return Post(body=body)
```

As you can see, this implementation chooses to only use the `body` attribute from the JSON dictionary. The `body_html` attribute is ignored since the server-side Markdown rendering is automatically triggered by an SQLAlchemy event whenever the `body` attribute is modified. The `timestamp` attribute does not need to be given unless the client is allowed to back- or future-date posts, which is not a feature this application supports. The `author_url` field is not used because the client has no authority to select the author of a blog post; the only possible value for this field is that of the authenticated user. The `comments_url` and `comment_count` attributes are automatically generated from a database relationship, so there is no useful information in them that is needed to create a `Post`. Finally, the `url` field is ignored because in this implementation the resource URLs are defined by the server, not the client.

Note how error checking is done. If the `body` field is missing or empty then a `ValidationError` exception is raised. Raising an exception is in this case the appropriate way to deal with the error because this method does not have enough knowledge to properly handle the error condition. The exception effectively passes the error up to the caller, enabling higher-level code to do the error handling. The `ValidationError` class is implemented as a simple subclass of Python's `ValueError`. This implementation is shown in Example 14-15.

Example 14-15. app/exceptions.py: ValidationError exception

```
class ValidationError(ValueError):
    pass
```

The application now needs to handle this exception by providing the appropriate response to the client. To avoid having to add exception-catching code in view functions, a global exception handler can be installed using Flask's `errorhandler` decorator. A handler for the `ValidationError` exception is shown in Example 14-16.

Example 14-16. app/api/errors.py: API error handler for ValidationError exceptions

```
@api.errorhandler(ValidationError)
def validation_error(e):
    return bad_request(e.args[0])
```

The `errorhandler` decorator is the same one that is used to register handlers for HTTP status codes, but in this usage it takes an `Exception` class as an argument. The decorated function will be invoked any time an exception of the given class is raised. Note that the decorator is obtained from the API blueprint, so this handler will be invoked only when the exception is raised while a route from the blueprint is being handled. Using this technique, the code in view functions can be written very cleanly and concisely, without the need to include error checking. For example:

```
@api.route('/posts/', methods=['POST'])
def new_post():
    post = Post.from_json(request.json)
    post.author = g.current_user
    db.session.add(post)
    db.session.commit()
    return jsonify(post.to_json())
```

Implementing Resource Endpoints

What remains is to implement the routes that handle the different resources. The `GET` requests are typically the easiest because they just return information and don't need to make any changes. Example 14-17 shows the two `GET` handlers for blog posts.

Example 14-17. app/api/posts.py: GET resource handlers for posts

```
@api.route('/posts/')
def get_posts():
    posts = Post.query.all()
    return jsonify({ 'posts': [post.to_json() for post in posts] })

@api.route('/posts/<int:id>')
def get_post(id):
    post = Post.query.get_or_404(id)
    return jsonify(post.to_json())
```

The first route handles the request for the collection of posts. This function uses a list comprehension to generate the JSON version of all the posts. The second route

returns a single blog post and responds with a code 404 error when the given `id` is not found in the database.

The `POST` handler for blog post resources inserts a new blog post in the database. This route is shown in Example 14-18.

Example 14-18. app/api/posts.py: POST resource handler for posts

```
@api.route('/posts/', methods=['POST'])
@permission_required(Permission.WRITE)
def new_post():
    post = Post.from_json(request.json)
    post.author = g.current_user
    db.session.add(post)
    db.session.commit()
    return jsonify(post.to_json()), 201, \
        {'Location': url_for('api.get_post', id=post.id)}
```

This view function is wrapped in a `permission_required` decorator (shown in an upcoming example) that ensures that the authenticated user has the permission to write blog posts. The actual creation of the blog post is straightforward due to the error handling support that was implemented previously. A blog post is created from the JSON data and its author is explicitly assigned as the authenticated user. After the model is written to the database, a 201 status code is returned and a `Location` header is added with the URL of the newly created resource.

Note that as a convenience to clients, the body of the response includes the new resource. This will save the client from having to issue a `GET` request for it immediately after creating the resource.

The `permission_required` decorator used to prevent unauthorized users from creating new blog posts is similar to the one used in the application but is customized for the API blueprint. The implementation is shown in Example 14-19.

Example 14-19. app/api/decorators.py: permission_required decorator

```
def permission_required(permission):
    def decorator(f):
        @wraps(f)
        def decorated_function(*args, **kwargs):
            if not g.current_user.can(permission):
                return forbidden('Insufficient permissions')
            return f(*args, **kwargs)
        return decorated_function
    return decorator
```

The PUT handler for blog posts, used for editing existing resources, is shown in Example 14-20.

Example 14-20. app/api/posts.py: PUT resource handler for posts

```
@api.route('/posts/<int:id>', methods=['PUT'])
@permission_required(Permission.WRITE)
def edit_post(id):
    post = Post.query.get_or_404(id)
    if g.current_user != post.author and \
            not g.current_user.can(Permission.ADMIN):
        return forbidden('Insufficient permissions')
    post.body = request.json.get('body', post.body)
    db.session.add(post)
    db.session.commit()
    return jsonify(post.to_json())
```

The permission checks are more complex in this case. The standard check for permission to write blog posts is done with the decorator, but to allow a user to edit a blog post the function must also ensure that the user is the author of the post or else is an administrator. This check is added explicitly to the view function. If this check had to be added in many view functions, building a decorator for it would be a good way to avoid code repetition.

Since the application does not allow deletion of posts, the handler for the DELETE request method does not need to be implemented.

The resource handlers for users and comments are implemented in a similar way. Table 14-3 lists the set of resources implemented for this application and the HTTP methods each supports. The complete implementation is available for you to study in the GitHub repository for this application.

Table 14-3. Flasky API resources

Resource URL	Method	Description
/users/<int:id>	GET	Return a user.
/users/<int:id>/posts/	GET	Return all the blog posts written by a user.
/users/<int:id>/timeline/	GET	Return all the blog posts followed by a user.
/posts/	GET	Return all the blog posts.
/posts/	POST	Create a new blog post.
/posts/<int:id>	GET	Return a blog post.
/posts/<int:id>	PUT	Modify a blog post.
/posts/<int:id>/comments/	GET	Return the comments on a blog post.
/posts/<int:id>/comments/	POST	Add a comment to a blog post.
/comments/	GET	Return all the comments.

Resource URL	Method	Description
/comments/<int:id>	GET	Return a comment.

Note that the resources that were implemented offer only a subset of the functionality that is available through the web application. The list of supported resources could be expanded if necessary, such as to expose followers, to enable comment moderation, and to implement any other features that an API client might need.

Pagination of Large Resource Collections

The `GET` requests that return a collection of resources can be extremely expensive and difficult to manage for very large collections. Like web applications, web services can choose to paginate collections.

Example 14-21 shows a possible implementation of pagination for the list of blog posts.

Example 14-21. app/api/posts.py: Post pagination

```
@api.route('/posts/')
def get_posts():
    page = request.args.get('page', 1, type=int)
    pagination = Post.query.paginate(
        page, per_page=current_app.config['FLASKY_POSTS_PER_PAGE'],
        error_out=False)
    posts = pagination.items
    prev = None
    if pagination.has_prev:
        prev = url_for('api.get_posts', page=page-1)
    next = None
    if pagination.has_next:
        next = url_for('api.get_posts', page=page+1)
    return jsonify({
        'posts': [post.to_json() for post in posts],
        'prev_url': prev,
        'next_url': next,
        'count': pagination.total
    })
```

The `posts` field in the JSON response contains the data items as before, but now it is just a page and not the complete set. The `prev_url` and `next_url` items contain the resource URLs for the previous and following pages, or `None` when a page in that direction is not available. The `count` value is the total number of items in the collection.

This technique can be applied to all the routes that return collections.

If you have cloned the application's Git repository on GitHub, you can run `git checkout 14a` to check out this version of the application. To ensure that you have all the dependencies installed, also run `pip install -r requirements/dev.txt`.

Testing Web Services with HTTPie

To test a web service, an HTTP client must be used. The two most used clients for testing Python web services from the command line are *cURL* and *HTTPie*. While both are useful tools, the latter has a much more concise and readable command line syntax that is tailored specifically to API requests. HTTPie is installed with *pip*:

```
(venv) $ pip install httpie
```

Assuming the development server is running on the default *http://127.0.0.1:5000* address, a `GET` request can be issued from another terminal window as follows:

```
(venv) $ http --json --auth <email>:<password> GET \
> http://127.0.0.1:5000/api/v1/posts
HTTP/1.0 200 OK
Content-Length: 7018
Content-Type: application/json
Date: Sun, 22 Dec 2013 08:11:24 GMT
Server: Werkzeug/0.9.4 Python/2.7.3

{
    "posts": [
        ...
    ],
    "prev_url": null
    "next_url": "http://127.0.0.1:5000/api/v1/posts/?page=2",
    "count": 150
}
```

Note the pagination links included in the response. Since this is the first page, a previous page is not defined, but a URL to obtain the next page and a total count were returned.

The following command sends a `POST` request to add a new blog post:

```
(venv) $ http --auth <email>:<password> --json POST \
> http://127.0.0.1:5000/api/v1/posts/ \
> "body=I'm adding a post from the *command line*."
HTTP/1.0 201 CREATED
Content-Length: 360
Content-Type: application/json
Date: Sun, 22 Dec 2013 08:30:27 GMT
Location: http://127.0.0.1:5000/api/v1/posts/111
Server: Werkzeug/0.9.4 Python/2.7.3

{
```

```
    "author": "http://127.0.0.1:5000/api/v1/users/1",
    "body": "I'm adding a post from the *command line*.",
    "body_html": "<p>I'm adding a post from the <em>command line</em>.</p>",
    "comments": "http://127.0.0.1:5000/api/v1/posts/111/comments",
    "comment_count": 0,
    "timestamp": "Sun, 22 Dec 2013 08:30:27 GMT",
    "url": "http://127.0.0.1:5000/api/v1/posts/111"
}
```

To use authentication tokens instead of a username and password, a `POST` request to */api/v1/tokens/* is sent first:

```
(venv) $ http --auth <email>:<password> --json POST \
> http://127.0.0.1:5000/api/v1/tokens/
HTTP/1.0 200 OK
Content-Length: 162
Content-Type: application/json
Date: Sat, 04 Jan 2014 08:38:47 GMT
Server: Werkzeug/0.9.4 Python/3.3.3

{
    "expiration": 3600,
    "token": "eyJpYXQiOjEzODg4MjQ3MjcsImV4cCI6MTM4ODgyODMyNywiYWxnIjoiSFMy..."
}
```

And now the returned token can be used to make calls into the API for the next hour by passing it along in the username field and leaving the password empty:

```
(venv) $ http --json --auth eyJpYXQ...: GET http://127.0.0.1:5000/api/v1/posts/
```

When the token expires, requests will be returned with a code 401 error, indicating that a new token needs to be obtained.

Congratulations! This chapter completes Part II, and with that the feature development phase of Flasky is complete. The next step is obviously to deploy it, and that brings a new set of challenges that are the subject of Part III.

PART III

The Last Mile

CHAPTER 15
Testing

There are two very good reasons for writing unit tests. When implementing new functionality, unit tests are used to confirm that the new code is working in the expected way. The same result can be obtained by testing manually, but of course automated tests save time and effort because they can be repeated easily.

A second, more important reason is that each time the application is modified, all the unit tests built around it can be executed to ensure that there are no *regressions* in the existing code; in other words, that the new changes did not affect the way the older code works.

Unit tests have been a part of Flasky since the very beginning, with tests designed to exercise specific features of the application implemented in the database model classes. These classes are easy to test outside of the context of a running application, so given that it takes little effort, implementing unit tests for all the features that exist in the database models is the best way to ensure at least that part of the application starts robust and stays that way.

This chapter discusses ways to improve and extend unit testing to other areas of the application.

Obtaining Code Coverage Reports

Having a test suite is important, but it is equally important to know how good or bad it is. Code coverage tools measure how much of the application is exercised by unit tests and can provide a detailed report that indicates which parts of the application code are not being tested. This information is invaluable, because it can be used to direct the effort of writing new tests to the areas that need it most.

Python has an excellent code coverage tool appropriately called *coverage*. You can install it with *pip*:

```
(venv) $ pip install coverage
```

This tool comes as a command-line script that can launch any Python application with code coverage enabled, but it also provides more convenient scripting access to start the coverage engine programmatically. To have coverage metrics nicely integrated into the `flask test` command added in Chapter 7, a `--coverage` option can be added. The implementation of this option is shown in Example 15-1.

Example 15-1. flasky.py: coverage metrics

```
import os
import sys
import click

COV = None
if os.environ.get('FLASK_COVERAGE'):
    import coverage
    COV = coverage.coverage(branch=True, include='app/*')
    COV.start()

# ...

@app.cli.command()
@click.option('--coverage/--no-coverage', default=False,
              help='Run tests under code coverage.')
def test(coverage):
    """Run the unit tests."""
    if coverage and not os.environ.get('FLASK_COVERAGE'):
        os.environ['FLASK_COVERAGE'] = '1'
        os.execvp(sys.executable, [sys.executable] + sys.argv)
    import unittest
    tests = unittest.TestLoader().discover('tests')
    unittest.TextTestRunner(verbosity=2).run(tests)
    if COV:
        COV.stop()
        COV.save()
        print('Coverage Summary:')
        COV.report()
        basedir = os.path.abspath(os.path.dirname(__file__))
        covdir = os.path.join(basedir, 'tmp/coverage')
        COV.html_report(directory=covdir)
        print('HTML version: file://%s/index.html' % covdir)
        COV.erase()
```

The code coverage support is enabled by passing the `--coverage` option to the `flask test` command. To add the Boolean option to the `test` custom command, the

`click.option` decorator is used. Click then passes the value of the Boolean flag as an argument to the function.

But integrating code coverage in the *flasky.py* script presents a small problem. By the time the `--coverage` option is received in the `test()` function, it is already too late to enable coverage metrics; by that time all the code in the global scope has already executed. So, to get accurate metrics, the script recursively restarts itself after setting the `FLASK_COVERAGE` environment variable. In the second run, the top of the script finds that the environment variable is set and turns on coverage from the start, even before all the application imports.

The `coverage.coverage()` function starts the coverage engine. The `branch=True` option enables branch coverage analysis, which, in addition to tracking which lines of code execute, checks whether for every conditional both the `True` and `False` cases have executed. The `include` option is used to limit coverage analysis to the files that are inside the application package, which is the only code that needs to be measured. Without the `include` option, all the extensions installed in the virtual environment and the code for the tests itself would be included in the coverage reports—and that would add a lot of noise to the report.

After all the tests have executed, the `test()` function writes a report to the console and also writes a nicer HTML report to disk. The HTML version shows all the source code annotated with colors that indicate the lines that are covered by the tests and the ones that are not.

> If you have cloned the application's Git repository on GitHub, you can run `git checkout 15a` to check out this version of the application. To ensure that you have all the dependencies installed, also run `pip install -r requirements/dev.txt`.

An example of the text-based report follows:

```
(venv) $ flask test --coverage
...
.----------------------------------------------------------------------
Ran 23 tests in 6.337s

OK
Coverage Summary:
Name                            Stmts   Miss Branch BrPart  Cover
-----------------------------------------------------------------
app/__init__.py                    32      0      0      0   100%
app/api_v1/__init__.py              3      0      0      0   100%
app/api_v1/authentication.py       29     18     10      0    28%
app/api_v1/comments.py             40     30     12      0    19%
app/api_v1/decorators.py           11      3      2      0    62%
```

```
app/api_v1/errors.py            17     10      0      0    41%
app/api_v1/posts.py             36     24      8      0    27%
app/api_v1/users.py             30     24     12      0    14%
app/auth/__init__.py             3      0      0      0   100%
app/auth/forms.py               45      8      8      0    70%
app/auth/views.py              116     91     42      0    16%
app/decorators.py               14      3      2      0    69%
app/email.py                    15      9      0      0    40%
app/exceptions.py                2      0      0      0   100%
app/main/__init__.py             6      1      0      0    83%
app/main/errors.py              20     15      6      0    19%
app/main/forms.py               39      7      6      0    71%
app/main/views.py              178    140     34      0    18%
app/models.py                  236     42     42      6    79%
-----------------------------------------------------------------
TOTAL                          872    425    184      6    45%
HTML version: file:///home/flask/flasky/tmp/coverage/index.html
```

The report shows an overall coverage of 45%, which is not terrible, but isn't very good either. The model classes, which have received all the unit testing attention so far, constitute a total of 236 statements, of which 79% are covered in tests. Obviously the *views.py* files in the `main` and `auth` blueprints and the routes in the `api_v1` blueprint all have very low coverage, since these are not exercised in any of the existing unit tests. And of course, these coverage metrics are not indicative of how much bug-free code exists in the project, since other factors (such as the quality of the tests) play a big role in that.

Armed with this report, it is easy to determine where tests need to be added to the test suite to improve coverage—but unfortunately, not all parts of the application can be tested as easily as the database models. The next two sections discuss more advanced testing strategies that can be applied to view functions, forms, and templates.

The Flask Test Client

Some portions of the application code rely heavily on the environment that is created by a running application. For example, you can't simply invoke the code in a view function to test it, since the function may need to access Flask context variables such as `request` or `session`, it may be expecting form data provided in a `POST` request, and it may also require a logged-in user. In short, view functions can run only within the context of a request and a running application.

Flask comes equipped with a *test client* to try to address this problem, at least to some extent. The test client replicates the environment that exists when an application is running inside a web server, allowing tests to act as clients and send requests.

The view functions do not see any major differences when executed under the test client; requests are received and routed to the appropriate view functions, from which

responses are generated and returned. After a view function executes, its response is passed to the test, which can check it for correctness.

Testing Web Applications

Example 15-2 shows a unit testing framework that uses the test client.

Example 15-2. tests/test_client.py: framework for tests using the Flask test client

```
import unittest
from app import create_app, db
from app.models import User, Role

class FlaskClientTestCase(unittest.TestCase):
    def setUp(self):
        self.app = create_app('testing')
        self.app_context = self.app.app_context()
        self.app_context.push()
        db.create_all()
        Role.insert_roles()
        self.client = self.app.test_client(use_cookies=True)

    def tearDown(self):
        db.session.remove()
        db.drop_all()
        self.app_context.pop()

    def test_home_page(self):
        response = self.client.get('/')
        self.assertEqual(response.status_code, 200)
        self.assertTrue('Stranger' in response.get_data(as_text=True))
```

Compared to *tests/test_basics.py*, this module adds a `self.client` instance variable, which is the Flask test client object. This object exposes methods that issue requests into the application. When the test client is created with the `use_cookies` option enabled, it will accept and send cookies in the same way browsers do, so functionality that relies on cookies to recall context between requests can be used. In particular, this approach enables the use of user sessions, which are stored in cookies.

The `test_home_page()` test is a simple example of what the test client can do. In this example, a request for the root URL of the application is issued. The return value of the `get()` method of the test client is a Flask response object containing the response returned by the invoked view function. To check whether the test was successful, the status code of the response is checked, and then the body of the response, obtained from `response.get_data()`, is searched for the word `"Stranger"`, which is part of the "Hello, Stranger!" greeting shown to anonymous users. Note that `get_data()`

returns the response body as a byte array by default; passing `as_text=True` converts it to a string, which is easier to work with.

The test client can also send `POST` requests that include form data using the `post()` method, but submitting forms presents a small complication. As discussed in Chapter 4, all forms generated by Flask-WTF have a hidden field with a CSRF token that needs to be submitted along with the form. To be able to send the CSRF token, a test would need to request the page that displays the form, then parse the HTML returned in that response and extract the token, so that it can then send it with the form data. To avoid the hassle of dealing with CSRF tokens in tests, it is better to disable CSRF protection in the testing configuration. This is shown in Example 15-3.

Example 15-3. config.py: disabling CSRF protection in the testing configuration

```
class TestingConfig(Config):
    #...
    WTF_CSRF_ENABLED = False
```

Example 15-4 shows a more advanced unit test that simulates a new user registering an account, logging in, confirming the account with a confirmation token, and finally logging out.

Example 15-4. tests/test_client.py: simulation of a new user workflow with the Flask test client

```
class FlaskClientTestCase(unittest.TestCase):
    # ...
    def test_register_and_login(self):
        # register a new account
        response = self.client.post('/auth/register', data={
            'email': 'john@example.com',
            'username': 'john',
            'password': 'cat',
            'password2': 'cat'
        })
        self.assertEqual(response.status_code, 302)

        # log in with the new account
        response = self.client.post('/auth/login', data={
            'email': 'john@example.com',
            'password': 'cat'
        }, follow_redirects=True)
        self.assertEqual(response.status_code, 200)
        self.assertTrue(re.search('Hello,\s+john!',
                                  response.get_data(as_text=True)))
        self.assertTrue(
            'You have not confirmed your account yet' in response.get_data(
                as_text=True))
```

```
        # send a confirmation token
        user = User.query.filter_by(email='john@example.com').first()
        token = user.generate_confirmation_token()
        response = self.client.get('/auth/confirm/{}'.format(token),
                                   follow_redirects=True)
        user.confirm(token)
        self.assertEqual(response.status_code, 200)
        self.assertTrue(
            'You have confirmed your account' in response.get_data(
                as_text=True))

        # log out
        response = self.client.get('/auth/logout', follow_redirects=True)
        self.assertEqual(response.status_code, 200)
        self.assertTrue('You have been logged out' in response.get_data(
            as_text=True))
```

The test begins with a form submission to the registration route. The `data` argument to `post()` is a dictionary with the form fields, which must exactly match the field names defined in the HTML form. Since CSRF protection is now disabled in the testing configuration, there is no need to send the CSRF token with the form.

The */auth/register* route can respond in two ways. If the registration data is valid, a redirect sends the user to the login page. In the case of an invalid registration, the response renders the page with the registration form again, including any appropriate error messages. To validate that the registration was accepted, the test checks that the status code of the response is 302, which is the code for a redirect.

The second section of the test issues a login request to the application using the email and password just registered. This is done with a `POST` request to the */auth/login* route. This time a `follow_redirects=True` argument is included in the `post()` call to make the test client work like a browser and automatically issue a `GET` request for the redirected URL. With this option, status code 302 will not be returned; instead, the response from the redirected URL is returned.

A successful response to the login submission would now have a page that greets the user by their username and then indicates that the account needs to be confirmed to gain access. Two assert statements verify that this is the page returned. Here, it is interesting to note that a search for the string `'Hello, john!'` would not work because this string is assembled from static and dynamic portions, so due to the way the Jinja2 template was created the final HTML has extra whitespace in between these two words. To avoid an error in this test due to the whitespace, a regular expression is used.

The next step is to confirm the account, which presents another small obstacle. The confirmation URL is sent to the user by email during registration, so there is no easy way to access it from the test. The solution presented in the test bypasses the token

that was generated as part of the registration and generates another one directly from the `User` instance. Another possibility would have been to extract the token by parsing the email body, which Flask-Mail saves when running in a testing configuration.

With the token at hand, the next step of the test is to simulate the user clicking the confirmation token URL received by email. This is achieved by sending a `GET` request to the confirmation URL, which includes the token. The response to this request is a redirect to the home page, but once again `follow_redirects=True` is specified, so the test client requests the redirected page automatically and returns it. The response is checked for the greeting and a flashed message that informs the user that the confirmation was successful.

The final step in this test is to send a `GET` request to the logout route; to confirm that this has worked, the test searches for the flashed message in the response.

> If you have cloned the application's Git repository on GitHub, you can run `git checkout 15b` to check out this version of the application.

Testing Web Services

The Flask test client can also be used to test RESTful web services. Example 15-5 shows an example unit test class with two tests.

Example 15-5. tests/test_api.py: RESTful API testing with the Flask test client

```
class APITestCase(unittest.TestCase):
    # ...
    def get_api_headers(self, username, password):
        return {
            'Authorization':
                'Basic ' + b64encode(
                    (username + ':' + password).encode('utf-8')).decode('utf-8'),
            'Accept': 'application/json',
            'Content-Type': 'application/json'
        }

    def test_no_auth(self):
        response = self.client.get(url_for('api.get_posts'),
                                   content_type='application/json')
        self.assertEqual(response.status_code, 401)

    def test_posts(self):
        # add a user
        r = Role.query.filter_by(name='User').first()
        self.assertIsNotNone(r)
```

```
        u = User(email='john@example.com', password='cat', confirmed=True,
                 role=r)
        db.session.add(u)
        db.session.commit()

        # write a post
        response = self.client.post(
            '/api/v1/posts/',
            headers=self.get_api_headers('john@example.com', 'cat'),
            data=json.dumps({'body': 'body of the *blog* post'}))
        self.assertEqual(response.status_code, 201)
        url = response.headers.get('Location')
        self.assertIsNotNone(url)

        # get the new post
        response = self.client.get(
            url,
            headers=self.get_api_headers('john@example.com', 'cat'))
        self.assertEqual(response.status_code, 200)
        json_response = json.loads(response.get_data(as_text=True))
        self.assertEqual('http://localhost' + json_response['url'], url)
        self.assertEqual(json_response['body'], 'body of the *blog* post')
        self.assertEqual(json_response['body_html'],
                         '<p>body of the <em>blog</em> post</p>')
```

The `setUp()` and `tearDown()` methods for testing the API are the same as for the regular application, but the cookie support does not need to be configured because the API does not use it. The `get_api_headers()` method is a helper method that returns the common headers that need to be sent with most API requests. These include the authentication credentials and the MIME type-related headers.

The `test_no_auth()` test is a simple test that ensures that a request that does not include authentication credentials is rejected with error code 401. The `test_posts()` test adds a user to the database and then uses the RESTful API to insert a blog post and then read it back. Any requests that send data in the body must encode it with `json.dumps()`, because the Flask test client does not automatically encode to JSON. Likewise, response bodies are also returned in JSON format and must be decoded with `json.loads()` before they can be inspected.

If you have cloned the application's Git repository on GitHub, you can run `git checkout 15c` to check out this version of the application.

End-to-End Testing with Selenium

The Flask test client cannot fully emulate the environment of a running application. For example, any application that relies on JavaScript code running in the client browser will not work, as the JavaScript code included in the responses will be returned to the test without being executed.

When tests require the complete environment, there is no other choice than to use a real web browser connected to the application running on a real web server. Fortunately, most web browsers can be automated. Selenium is a web browser automation tool that supports the most popular web browsers in the three major operating systems.

The Python interface for Selenium is installed with *pip*:

```
(venv) $ pip install selenium
```

Selenium requires a driver for the desired web browser to be installed separately, in addition to the browser itself. There are drivers for all major web browsers, so an application could set up a sophisticated framework to test several browsers. For this application, however, only the Google Chrome web browser will be used for automated tests, with its corresponding driver, *ChromeDriver*. If you are using a macOS computer with the *brew* package installer, you can install ChromeDriver as follows:

```
(venv) $ brew install chromedriver
```

For Linux, Microsoft Windows, or a macOS computer without *brew*, you can download a regular ChromeDriver installer from the ChromeDriver website.

Testing with Selenium requires the application to be running inside a web server that is listening for real HTTP requests. The method that will be shown in this section starts the application with the development server in a background thread while the tests run on the main thread. Under the control of the tests, Selenium launches a web browser and makes it connect to the application to perform the required operations.

A problem with this approach is that after all the tests have completed, the Flask server needs to be stopped, ideally in a graceful way, so that background tasks such as the code coverage engine can cleanly complete their work. The Werkzeug web server has a shutdown option, but because the server is running isolated in its own thread, the only way to ask the server to shut down is by sending a regular HTTP request. Example 15-6 shows the implementation of a server shutdown route.

Example 15-6. _app/main/views.py: server shutdown route

```
@main.route('/shutdown')
def server_shutdown():
    if not current_app.testing:
        abort(404)
    shutdown = request.environ.get('werkzeug.server.shutdown')
    if not shutdown:
        abort(500)
    shutdown()
    return 'Shutting down...'
```

The shutdown route will work only when the application is running in testing mode; invoking it in other configurations will return a 404 status code response. The actual shutdown procedure involves calling a shutdown function that Werkzeug exposes in the environment. After calling this function and returning from the request, the development web server will know that it needs to exit gracefully.

Example 15-7 shows the layout of a test case that is configured to run tests with Selenium.

Example 15-7. tests/test_selenium.py: framework for tests using Selenium

```
from selenium import webdriver

class SeleniumTestCase(unittest.TestCase):
    client = None

    @classmethod
    def setUpClass(cls):
        # start Chrome
        options = webdriver.ChromeOptions()
        options.add_argument('headless')
        try:
            cls.client = webdriver.Chrome(chrome_options=options)
        except:
            pass

        # skip these tests if the browser could not be started
        if cls.client:
            # create the application
            cls.app = create_app('testing')
            cls.app_context = cls.app.app_context()
            cls.app_context.push()

            # suppress logging to keep unittest output clean
            import logging
            logger = logging.getLogger('werkzeug')
            logger.setLevel("ERROR")
```

```
        # create the database and populate with some fake data
        db.create_all()
        Role.insert_roles()
        fake.users(10)
        fake.posts(10)

        # add an administrator user
        admin_role = Role.query.filter_by(permissions=0xff).first()
        admin = User(email='john@example.com',
                     username='john', password='cat',
                     role=admin_role, confirmed=True)
        db.session.add(admin)
        db.session.commit()

        # start the Flask server in a thread
        cls.server_thread = threading.Thread(
            target=cls.app.run, kwargs={'debug': 'false',
                                        'use_reloader': False,
                                        'use_debugger': False})
        cls.server_thread.start()

    @classmethod
    def tearDownClass(cls):
        if cls.client:
            # stop the Flask server and the browser
            cls.client.get('http://localhost:5000/shutdown')
            cls.client.quit()
            cls.server_thread.join()

            # destroy database
            db.drop_all()
            db.session.remove()

            # remove application context
            cls.app_context.pop()

    def setUp(self):
        if not self.client:
            self.skipTest('Web browser not available')

    def tearDown(self):
        pass
```

The `setUpClass()` and `tearDownClass()` class methods are invoked before and after the tests in this class execute. The setup involves starting an instance of Chrome through Selenium's `webdriver` API, and creating an application and a database with some initial fake data for tests to use. The application is started in a thread using the `app.run()` method. At the end the application receives a request to */shutdown*, which causes the background thread to end. The browser is then closed and the test database removed.

Before the Flask command-line interface based on Click was introduced, you had to start the Flask development web server by calling `app.run()` from the application's main script, or else use a third-party extension such as Flask-Script. While using `app.run()` to start a server is now replaced with the `flask run` command, the `app.run()` method continues to be supported, and here you can see how it can still be useful for complex unit testing situations.

Selenium supports many other web browsers besides Chrome. Consult the Selenium documentation if you wish to use another web browser or test additional browsers.

The `setUp()` method that runs before each test skips tests if Selenium cannot start the web browser in the `startUpClass()` method. In Example 15-8 you can see an example test built with Selenium.

Example 15-8. tests/test_selenium.py: example Selenium unit test

```
class SeleniumTestCase(unittest.TestCase):
    # ...

    def test_admin_home_page(self):
        # navigate to home page
        self.client.get('http://localhost:5000/')
        self.assertTrue(re.search('Hello,\s+Stranger!',
                                  self.client.page_source))

        # navigate to login page
        self.client.find_element_by_link_text('Log In').click()
        self.assertIn('<h1>Login</h1>', self.client.page_source)

        # log in
        self.client.find_element_by_name('email').\
            send_keys('john@example.com')
        self.client.find_element_by_name('password').send_keys('cat')
        self.client.find_element_by_name('submit').click()
        self.assertTrue(re.search('Hello,\s+john!', self.client.page_source))

        # navigate to the user's profile page
        self.client.find_element_by_link_text('Profile').click()
        self.assertIn('<h1>john</h1>', self.client.page_source)
```

This test logs in to the application using the administrator account that was created in `setUpClass()` and then opens the user's profile page. Note how different the testing methodology is from the Flask test client. When testing with Selenium, tests send commands to the web browser and never interact with the application directly. The

commands closely match the actions that a real user would perform with a mouse or keyboard.

The test begins with a call to `get()` with the home page of the application. In the browser, this causes the URL to be entered in the address bar. To verify this step, the page source is checked for the "Hello, Stranger!" greeting.

To go to the sign-in page, the test looks for the "Log In" link using `find_element_by_link_text()` and then calls `click()` on it to trigger a real click in the browser. Selenium provides several `find_element_by...()` convenience methods that can search for elements within the HTML page in different ways.

To log in to the application, the test locates the email and password form fields by their names using `find_element_by_name()` and then writes text into them with `send_keys()`. The form is submitted by calling `click()` on the submit button. The personalized greeting is checked to ensure that the login was successful and the browser is now on the home page.

The final part of the test locates the "Profile" link in the navigation bar and clicks it. To verify that the profile page was loaded, the heading with the username is searched in the page source.

> If you have cloned the application's Git repository on GitHub, you can run `git checkout 15d` to check out this version of the application. This update contains a database migration, so remember to run `flask db upgrade` after you check out the code. To ensure that you have all the dependencies installed, also run `pip install -r requirements/dev.txt`.

When you run the unit tests with the `flask test` command there will be no visible difference. The `test_admin_home_page` unit test in Example 15-8 will run a headless Chrome instance and perform all the actions on it. If you want to see the actions performed in a real Chrome window, comment out the line `options.add_argument('headless')` in the `setUpClass()` method, so that Selenium creates a regular Chrome window.

Is It Worth It?

By now you may be asking yourself if testing using the Flask test client or Selenium is really worth the trouble. It is a valid question, and it does not have a simple answer.

Whether you like it or not, your application will be tested. If you don't test it yourself, then your users will become the unwilling testers; they will find the bugs, and then you will have to fix them under pressure. Simple and focused tests like the ones that exercise database models and other parts of the application that can be executed out-

side of the context of an application should always be carried out, as they have a very low cost and ensure the proper functioning of the core pieces of application logic.

End-to-end tests of the type that the Flask test client and Selenium can carry out are sometimes necessary, but due to the increased complexity of writing them, they should be used only for functionality that cannot be tested in isolation. The application code should be organized so that it is possible to push the business logic into application modules that are independent of the context of the application, and thus can be tested more easily. The code that exists in view functions should be simple and just act as a thin layer that accepts requests and invokes the corresponding actions in other classes or functions that encapsulate the application logic.

So yes, testing is absolutely worth it. But it is important to design an efficient testing strategy and write code that can take advantage of it.

CHAPTER 16
Performance

Nobody likes slow applications. Long waits for pages to load frustrate users, so it is important to detect and correct performance problems as soon as they appear. In this chapter, two important performance aspects of web applications are considered.

Logging Slow Database Performance

When application performance slowly degenerates with time, it is likely due to slow database queries, which get worse as the size of the database grows. Optimizing database queries can be as simple as adding more indexes or as complex as adding a cache between the application and the database. The `explain` statement, available in most database query languages, shows the steps the database takes to execute a given query, often exposing inefficiencies in database or index design.

But before starting to optimize queries, it is necessary to determine which queries are the ones that are worth optimizing. During a typical request several database queries may be issued, so it is often hard to identify which of all the queries are the slow ones. Flask-SQLAlchemy has an option to record statistics about database queries issued during a request. In Example 16-1 you can see how this feature can be used to *log* queries that are slower than a configured threshold.

Example 16-1. app/main/views.py: reporting slow database queries

```
from flask_sqlalchemy import get_debug_queries

@main.after_app_request
def after_request(response):
    for query in get_debug_queries():
        if query.duration >= current_app.config['FLASKY_SLOW_DB_QUERY_TIME']:
            current_app.logger.warning(
```

```
            'Slow query: %s\nParameters: %s\nDuration: %fs\nContext: %s\n' %
                (query.statement, query.parameters, query.duration,
                 query.context))
    return response
```

This functionality is attached to an after_app_request handler, which works in a similar way to the before_app_request handler but is invoked after the view function that handles the request returns. Flask passes the response object to the after_app_request handler in case it needs to be modified.

In this case, the after_app_request handler does not modify the response; it just gets the query timings recorded by Flask-SQLAlchemy and then logs the slow ones to the application logger that Flask sets up at app.logger, before returning the response, which will then be sent to the client.

The get_debug_queries() function returns the queries issued during the request as a list. The information provided for each query is shown in Table 16-1.

Table 16-1. Query statistics recorded by Flask-SQLAlchemy

Name	Description
statement	The SQL statement
parameters	The parameters used with the SQL statement
start_time	The time the query was issued
end_time	The time the query returned
duration	The duration of the query in seconds
context	A string that indicates the source code location where the query was issued

The after_app_request handler walks the list and logs any queries that lasted longer than a threshold given in the configuration variable FLASKY_SLOW_DB_QUERY_TIME. The logging is issued at the warning level in this application, but in some cases it may make sense to treat slow database alerts as errors.

The get_debug_queries() function is enabled only in debug mode by default. Unfortunately, database performance problems rarely show up during development because much smaller databases are used. For this reason, it is much more useful to enable this option in production. Example 16-2 shows the configuration changes that are necessary to enable database query performance monitoring in production mode.

Example 16-2. config.py: configuration for slow query reporting

```
class Config:
    # ...
    SQLALCHEMY_RECORD_QUERIES = True
    FLASKY_SLOW_DB_QUERY_TIME = 0.5
    # ...
```

`SQLALCHEMY_RECORD_QUERIES` tells Flask-SQLAlchemy to enable the recording of query statistics. The slow query threshold is set to half a second. Both configuration variables were included in the base `Config` class, so they will be enabled for all configurations.

Whenever a slow query is detected, an entry will be written to Flask's application logger. To be able to store these log entries, the logger must be configured. The logging configuration largely depends on the platform that hosts the application. Some examples are shown in Chapter 17.

> If you have cloned the application's Git repository on GitHub, you can run `git checkout 16a` to check out this version of the application.

Source Code Profiling

Another possible source of performance problems is high CPU consumption, caused by functions that perform heavy computing. Source code profilers are useful in finding the slowest parts of an application. A profiler watches a running application and records the functions that are called and how long each takes to run. It then produces a detailed report showing the slowest functions.

> Profiling is typically done only in a development environment. A source code profiler makes the application run much slower than normal, because it has to observe and take notes on all that is happening in real time. Profiling on a production system is not recommended, unless a lightweight profiler specifically designed to run in a production environment is used.

Flask's development web server, which comes from Werkzeug, can optionally enable the Python profiler for each request. Example 16-3 adds a new command-line option to the application that starts the web server under the profiler.

Example 16-3. flasky.py: running the application under the request profiler

```
@app.cli.command()
@click.option('--length', default=25,
              help='Number of functions to include in the profiler report.')
@click.option('--profile-dir', default=None,
              help='Directory where profiler data files are saved.')
def profile(length, profile_dir):
    """Start the application under the code profiler."""
    from werkzeug.contrib.profiler import ProfilerMiddleware
    app.wsgi_app = ProfilerMiddleware(app.wsgi_app, restrictions=[length],
                                      profile_dir=profile_dir)
    app.run(debug=False)
```

This command attaches the `ProfilerMiddleware` from Werkzeug to the application, through its `wsgi_app` attribute. WSGI middlewares are invoked each time the web server dispatches a request to the application and can modify the way the request is handled, in this case by capturing profiling data. Note that the application is then started programmatically using the `app.run()` method.

If you have cloned the application's Git repository on GitHub, you can run `git checkout 16b` to check out this version of the application.

When the application is started with `flask profile`, the console will show the profiler statistics for each request, which will include the slowest 25 functions. The `--length` option can be used to change the number of functions shown in the report. If the `--profile-dir` option is given, the profile data for each request is saved to a file in the given directory. The profiler data files can be used to generate more detailed reports that include a *call graph*. For more information on the Python profiler, consult the official documentation.

The preparations for deployment are complete. The next chapter will give you an overview of what to expect when deploying your application.

CHAPTER 17

Deployment

The web development server that comes bundled with Flask is not robust, secure, or efficient enough to work in a production environment. In this chapter, production deployment options for Flask applications are examined.

Deployment Workflow

Regardless of the hosting method used, there are a series of tasks that must be carried out when the application is installed on a production server. These include the creation or update of the database tables.

Having to run these tasks manually each time the application is installed or upgraded is error prone and time consuming. Instead, a command that performs all the required tasks can be added to *flasky.py*.

Example 17-1 shows a `deploy` command implementation that is appropriate for Flasky.

Example 17-1. flasky.py: deploy command

```
from flask_migrate import upgrade
from app.models import Role, User

@manager.command
def deploy():
    """Run deployment tasks."""
    # migrate database to latest revision
    upgrade()

    # create or update user roles
    Role.insert_roles()
```

```
    # ensure all users are following themselves
    User.add_self_follows()
```

The functions invoked by this command were all created before; they are just invoked all together from a single command to simplify the deployment of the application.

> If you have cloned the application's Git repository on GitHub, you can run `git checkout 17a` to check out this version of the application.

These functions are all designed in a way that causes no problems if they are executed multiple times. Designing update functions in this way makes it possible to run just this `deploy` command every time an installation or upgrade is done without having to worry about side effects caused by a function that runs at the wrong time.

Logging of Errors During Production

When the application is running in debug mode, Werkzeug's interactive debugger appears whenever an error occurs. The *stack trace* of the error is displayed on the web page, and it is possible to look at the source code and even evaluate expressions in the context of each stack frame using Flask's interactive web-based debugger.

The debugger is an excellent tool to debug application problems during development, but obviously it cannot be used in a production deployment. Errors that occur in production are silenced and instead the user receives a discrete code 500 error page. But luckily, the stack traces of these errors are not completely lost, as Flask writes them to a log file.

During startup, Flask creates an instance of Python's `logging.Logger` class and attaches it to the application instance as `app.logger`. In debug mode, this logger writes to the console, but in production mode there are no handlers configured for it by default. Unless a handler is added, logs are not stored. The changes in Example 17-2 configure a logging handler that sends the errors that occur while running under the production configuration to the administrator email address configured in the `FLASKY_ADMIN` setting.

Example 17-2. config.py: sending email for application errors

```
class ProductionConfig(Config):
    # ...
    @classmethod
    def init_app(cls, app):
        Config.init_app(app)
```

```
# email errors to the administrators
import logging
from logging.handlers import SMTPHandler
credentials = None
secure = None
if getattr(cls, 'MAIL_USERNAME', None) is not None:
    credentials = (cls.MAIL_USERNAME, cls.MAIL_PASSWORD)
    if getattr(cls, 'MAIL_USE_TLS', None):
        secure = ()
mail_handler = SMTPHandler(
    mailhost=(cls.MAIL_SERVER, cls.MAIL_PORT),
    fromaddr=cls.FLASKY_MAIL_SENDER,
    toaddrs=[cls.FLASKY_ADMIN],
    subject=cls.FLASKY_MAIL_SUBJECT_PREFIX + ' Application Error',
    credentials=credentials,
    secure=secure)
mail_handler.setLevel(logging.ERROR)
app.logger.addHandler(mail_handler)
```

Recall that all configuration classes have an `init_app()` static method that is invoked by `create_app()`, which so far has not been used. In the implementation of this method for the `ProductionConfig` class, the application logger is now configured with a log handler that sends errors to an email recipient.

The logging level of the email logger is set to `logging.ERROR`, so only severe problems are going to be emailed. Messages logged on lesser levels can be logged to a file, syslog, or any other supported destination by adding the proper logging handlers. The logging method to use for these messages largely depends on the hosting platform.

If you have cloned the application's Git repository on GitHub, you can run `git checkout 17b` to check out this version of the application.

Cloud Deployment

The trend in application hosting is to host "in the cloud," but this can mean many different things. At the most basic level, cloud hosting can mean that the application is installed on one or more *virtual servers*, which for all intents and purposes operate and feel like physical machines, but in reality are virtual machines managed by the cloud operator. An example of these types of servers are those available through the *EC2* service from Amazon Web Services (AWS). Deploying an application to a virtual server is similar to doing a traditional deployment to a dedicated server, as described later in this chapter.

A more advanced deployment model is based on *containers*. A container isolates an application in an *image* of the application and its environment. A container image includes the application plus all the dependencies it needs to run. A container platform, such as Docker, can then install and execute a pregenerated container image on any system in which it runs.

Another deployment option, formally known as Platform as a Service (PaaS), frees the application developer from the mundane tasks of installing and maintaining the hardware and software platforms on which the application runs. In the PaaS model, a service provider offers a fully managed platform on which applications can run. All the application developer needs to do is upload the application code to the servers maintained by the provider, after which it automatically becomes available, usually within seconds. Most PaaS providers offer ways to dynamically "scale" the application by adding or removing servers as necessary to keep up with the number of requests received.

The remainder of this chapter offers an introduction to Heroku (one of the most popular PaaS providers), Docker containers, and finally traditional deployments, which are suitable for dedicated or virtual servers.

The Heroku Platform

Heroku was one of the first PaaS providers, having been in business since 2007. The Heroku platform is very flexible and supports a long list of programming languages, including Python. To deploy an application to Heroku, the developer uses Git to push the application to Heroku's special Git server, which automatically triggers the installation, upgrade, configuration, and deployment of the application.

Heroku uses units of computing called *dynos* to measure usage and charge for the service. The most common type of dyno is the *web dyno*, which represents a web server instance. An application can increase its request handling capacity by deploying more web dynos, each running an instance of the application. Another type of dyno is the *worker dyno*, which is used to perform background jobs or other support tasks.

The platform provides a large number of plug-ins and add-ons for databases, email support, and many other services. The following sections expand on some of the details involved in deploying Flasky to Heroku.

Preparing the Application

To work with Heroku, the application must be hosted in a Git repository. If you are working with an application that is hosted on a remote Git server, such as GitHub or Bitbucket, cloning the application will create a local Git repository that is perfect to

use with Heroku. If the application isn't already hosted in a Git repository, you'll need to create one for it on your development machine.

> If you plan on hosting your application on Heroku, it is a good idea to start using Git from the very beginning. GitHub has installation and setup guides for the three major operating systems in its help guide.

Creating a Heroku account

You must create an account with Heroku before you can use the service. Heroku provides a free tier that allows you to host a few simple applications, so this is a great platform to experiment with.

Installing the Heroku CLI

To work with the Heroku service, the Heroku CLI must be installed. This is a command-line client that manages the interactions with the service. Heroku provides installers for the three major operating systems.

The first thing to do after installing the CLI is to authenticate with your Heroku account through the `heroku login` command:

```
$ heroku login
Enter your Heroku credentials.
Email: <your-email-address>
Password: <your-password>
```

> It is important that your SSH public key is uploaded to Heroku, as this is what enables the `git push` command. Normally the `login` command creates and uploads an SSH public key automatically, but the `heroku keys:add` command can be used to upload your public key separately from the `login` command or if you need to upload additional keys.

Creating an application

The next step is to create an application. Before this is done, the application needs to be under Git source control. If you have been using the GitHub repository to follow along with the code in this book, then you already have a Git repository. If not, you will need to create one now. To register the application with Heroku, run the following command from the application's top-level directory:

```
$ heroku create <appname>
Creating <appname>... done
https://<appname>.herokuapp.com/ | https://git.heroku.com/<appname>.git
```

Heroku application names must be unique across all customers, so you need to think of a name that is not taken by any other application. As indicated by the output of the `create` command, once deployed the application will be available at *https://<appname>.herokuapp.com*. Heroku also supports using a custom domain name for your application.

As part of the application creation, Heroku creates a Git server dedicated to your application at *https://git.heroku.com/<appname>.git*. The `create` command adds this server to your local Git repository as a `git remote` with the name `heroku`:

```
$ git remote show heroku
* remote heroku
  Fetch URL: https://git.heroku.com/<appname>.git
  Push  URL: https://git.heroku.com/<appname>.git
  HEAD branch: (unknown)
```

The `flask` command requires the `FLASK_APP` environment variable to be set to work. To make sure that any commands that are executed in the Heroku environment succeed, it is a good idea to register this environment variable so that it is always set when Heroku executes commands related to this application. This can be done with the `config` command:

```
$ heroku config:set FLASK_APP=flasky.py
Setting FLASK_APP and restarting <appname>... done, v4
FLASK_APP: flasky.py
```

Provisioning a database

Heroku supports Postgres databases as an add-on. The free service tier includes a small database of up to 10,000 rows. To attach a Postgres database to your application, use the following command:

```
$ heroku addons:create heroku-postgresql:hobby-dev
Creating heroku-postgresql:hobby-dev on <appname>... free
Database has been created and is available
 ! This database is empty. If upgrading, you can transfer
 ! data from another database with pg:copy
Created postgresql-cubic-41298 as DATABASE_URL
Use heroku addons:docs heroku-postgresql to view documentation
```

As indicated by the output of the command, once the application runs inside the Heroku platform, it will see the database location and credentials in the `DATABASE_URL` environment variable. The format of this variable is a URL, exactly in the format SQLAlchemy expects. Recall that the *config.py* script uses the value of `DATABASE_URL` if it is defined, so the connection to the Postgres database will work automatically.

Configuring logging

Logging of fatal errors by email was added earlier, but in addition to that it is important to configure logging of lesser message categories. A good example of these types of messages are the warnings for slow database queries added in Chapter 16.

Heroku considers any output written by the application to `stdout` or `stderr` logs, so a logging handler needs to be added to generate this output. The logging output is captured by Heroku and made accessible through the Heroku client with the `heroku logs` command.

The logging configuration can be added to the `ProductionConfig` class in its `init_app()` static method. But since this type of logging is specific to Heroku, a better approach is to define a new configuration specifically for this platform, leaving `ProductionConfig` as a baseline configuration for different types of production platforms. The `HerokuConfig` class is shown in Example 17-3.

Example 17-3. config.py: Heroku configuration

```
class HerokuConfig(ProductionConfig):
    @classmethod
    def init_app(cls, app):
        ProductionConfig.init_app(app)

        # log to stderr
        import logging
        from logging import StreamHandler
        file_handler = StreamHandler()
        file_handler.setLevel(logging.INFO)
        app.logger.addHandler(file_handler)
```

When the application is executed by Heroku, it needs to know that this new configuration needs to be used. The application instance created in *flasky.py* uses the `FLASK_CONFIG` environment variable to know what configuration to use, so this variable needs to be set appropriately in the Heroku environment. Environment variables for the Heroku environment are set using the Heroku client's `config:set` command:

```
$ heroku config:set FLASK_CONFIG=heroku
Setting FLASK_CONFIG and restarting <appname>... done, v4
FLASK_CONFIG: heroku
```

To increase the security of your application, it is a good idea to configure a difficult-to-guess string as the application's secret key, which is used to sign the user session and the authentication tokens. The `Config` base class includes the `SECRET_KEY` attribute for this purpose, and sets its value from an environment variable of the same name if it exists. When working on the application in your development system it is okay to leave this variable undefined and let the `Config` class configure a hardcoded

value, but on a production platform it is extremely important to set a strong secret key that is not known to anyone, since a leaked key will enable an attacker to forge the contents of the user session or generate valid tokens. To make your key secure, just set the SECRET_KEY environment variable to a unique string that is not stored anywhere:

```
$ heroku config:set SECRET_KEY=d68653675379485599f7876a3b469a57
Setting SECRET_KEY and restarting <appname>... done, v4
SECRET_KEY: d68653675379485599f7876a3b469a57
```

There are many ways to generate random strings that are appropriate to be used as secret keys. You can do so with Python as follows:

```
(venv) $ python -c "import uuid; print(uuid.uuid4().hex)"
d68653675379485599f7876a3b469a57
```

Configuring email

Heroku does not provide an SMTP server, so an external server must be configured. There are several third-party add-ons that integrate production-ready email sending support with Heroku, but for testing and evaluation purposes it is sufficient to use the default Gmail configuration inherited from the base `Config` class.

Because it can be a security risk to embed login credentials directly in the script, the username and password to access the Gmail SMTP server are provided as environment variables (if you haven't yet, it is a very good idea that instead of using your personal email account you create a secondary email to use for testing):

```
$ heroku config:set MAIL_USERNAME=<your-gmail-username>
$ heroku config:set MAIL_PASSWORD=<your-gmail-password>
```

Adding a top-level requirements file

Heroku installs package dependencies from a *requirements.txt* file stored in the top-level directory of the application. All the dependencies in this file will be imported into a virtual environment managed by Heroku as part of the deployment.

The Heroku requirements file must include all the common requirements for the production version of the application, plus the `psycopg2` package that enables SQLAlchemy to access the Postgres database. A *heroku.txt* file with these dependencies can be added in the *requirements* directory and then imported from the top-level *requirements.txt* file as shown in Example 17-4.

Example 17-4. requirements.txt: Heroku requirements file

```
-r requirements/heroku.txt
```

Enabling Secure HTTP with Flask-SSLify

When the user logs in to the application by submitting a username and a password in a web form, these values are at risk of being intercepted by a malicious third party, as discussed several times before. During development this is not a problem, but this risk needs to be eliminated when you deploy the application on a production server. To prevent user credentials from being exposed while in transit, it is necessary to use secure HTTP, which encrypts all the communications between clients and the server using public key cryptography.

Heroku makes all applications that are accessed on the *herokuapp.com* domain available on both *http://* and *https://* without any configuration required. Because the application runs on Heroku's domain, it will use Heroku's own SSL certificate. The only necessary action to fully secure the application is to intercept any requests sent to the *http://* interface and redirect them to *https://*, which is exactly what the Flask-SSLify extension does.

As usual, Flask-SSLify is installed with *pip*:

```
(venv) $ pip install flask-sslify
```

The code that activates this extension is added to the application factory function, as shown in Example 17-5.

Example 17-5. app/__init__.py: redirecting all requests to secure HTTP

```python
def create_app(config_name):
    # ...
    if app.config['SSL_REDIRECT']:
        from flask_sslify import SSLify
        sslify = SSLify(app)
    # ...
```

Support for SSL needs to be enabled only in production mode, and only when the platform supports it. To make it easy to switch SSL on and off, a new configuration variable called `SSL_REDIRECT` is added. The base `Config` class sets it to `False`, so that SSL redirects are not used by default, and the class `HerokuConfig` overrides it so that only on that configuration are the redirects issued. The implementation of this configuration variable is shown in Example 17-6.

Example 17-6. config.py: configuring the use of SSL

```
class Config:
    # ...
    SSL_REDIRECT = False

class HerokuConfig(ProductionConfig):
    # ...
    SSL_REDIRECT = True if os.environ.get('DYNO') else False
```

The value of `SSL_REDIRECT` in `HerokuConfig` is only set to `True` if the environment variable `DYNO` exists. This variable is set by Heroku in its environment, so using the Heroku configuration for local testing does not activate the SSL redirects.

With these changes, the users will be forced to use the SSL server when accessing the application on Heroku—but there is one more detail that needs to be handled to make this support complete. When using Heroku, clients do not connect to the application directly but to a *reverse proxy server*. The reverse proxy server receives requests from many applications, and forwards them to each of them as appropriate. In this type of setup, only the proxy server runs in SSL mode; the SSL connection is *terminated* at the proxy server, and applications receive the forwarded requests from the proxy server without encryption. This presents a problem when the application needs to generate absolute URLs, because in the Flask application the request object describes the forwarded request, which is not encrypted, and not the original request sent by the client through an encrypted connection.

An example of the problem this can cause is with the generation of account confirmation or password reset links that are sent by email to users. When `url_for()` is called with `_external=True` to generate an absolute URL for these links, Flask will use *http://* for them, because it does not know that there is a reverse proxy that is accepting encrypted connections from the outside.

Proxy servers pass information that describes the original request from the client to the redirected web servers through custom HTTP headers, so it is possible to determine whether the user is communicating with the application over SSL by looking at these headers. Werkzeug provides a WSGI middleware that checks the custom headers from the proxy server and updates the request object accordingly so that, for example, `request.is_secure` reflects the encryption state of the request that the client sent to the reverse proxy server and not the request that the proxy server then forwarded to the application. Example 17-7 shows how to add the `ProxyFix` middleware to the application.

Example 17-7. config.py: adding support for proxy servers

```
class HerokuConfig(ProductionConfig):
    # ...
    @classmethod
    def init_app(cls, app):
        # ...

        # handle reverse proxy server headers
        from werkzeug.contrib.fixers import ProxyFix
        app.wsgi_app = ProxyFix(app.wsgi_app)
```

The middleware is added in the initialization method for the Heroku configuration. WSGI middlewares such as `ProxyFix` are added by wrapping the WSGI application. When a request comes, the middlewares get a chance to inspect the environment and make changes before the request is processed. The `ProxyFix` middleware is necessary not only for Heroku but in any deployment that uses a reverse proxy server.

Running a production web server

Heroku expects applications to start their own production web server and configure it to listen to requests on the port number set in the environment variable `PORT`.

The development web server that comes with Flask will perform very poorly in this situation because it is not designed to run in a production environment. Two production-ready web servers that work well with Flask applications are Gunicorn and uWSGI.

It is a good idea to install the chosen web server in the local virtual environment, so that it can be tested in a way similar to how it will run in the Heroku environment. For example, Gunicorn is installed as follows:

```
(venv) $ pip install gunicorn
```

To run the application locally under Gunicorn, use the following command:

```
(venv) $ gunicorn flasky:app
[2017-08-03 23:54:36 -0700] [INFO] Starting gunicorn 19.7.1
[2017-08-03 23:54:36 -0700] [INFO] Listening at: http://127.0.0.1:8000 (68982)
[2017-08-03 23:54:36 -0700] [INFO] Using worker: sync
[2017-08-03 23:54:36 -0700] [INFO] Booting worker with pid: 68985
```

The `flasky:app` argument tells Gunicorn where the application instance is located. The name given before the colon is the package or module that defines this instance, while the name after the colon is the actual application instance name. Note that Gunicorn uses port 8000 by default, not 5000 like Flask. Like the Flask development web server, you can exit Gunicorn with Ctrl+C.

The Gunicorn web server does not work on Microsoft Windows. The other recommended web server, uWSGI, does work on Windows, but it can be difficult to install due to it being written in native code. If you want to test the Heroku deployment on your Windows system, you can use Waitress, which is another pure Python web server that is in many ways similar to Gunicorn but has the advantage that it fully supports Windows. Waitress is installed with *pip*:

```
(venv) $ pip install waitress
```

To start the Waitress web server, use the `waitress-serve` command:

```
(venv) $ waitress-serve --port 8000 flasky:app
```

Adding a Procfile

Heroku needs to know what command to use to start the application. This command is given in a special file called *Procfile*. This file must be included in the top-level directory of the application.

Example 17-8 shows the contents of this file.

Example 17-8. Procfile: Heroku Procfile

```
web: gunicorn flasky:app
```

The format for the *Procfile* is very simple: in each line a task name is given, followed by a colon and then the command that runs the task. The task name `web` is special; it is recognized by Heroku as the task that starts the web server. Heroku will give this task a `PORT` environment variable set to the port on which the application needs to listen for requests. Gunicorn by default honors the `PORT` variable if it is set in the environment, so there is no need to include it in the startup command.

If you are using Microsoft Windows, or need your application to be fully compatible with that platform, you can instead use the Waitress web server:

```
web: waitress-serve --port=$PORT flasky:app
```

Applications can declare additional tasks with names other than `web` in the *Procfile*. Each task included in the *Procfile* will be started on its own dyno.

If you have cloned the application's Git repository on GitHub, you can run `git checkout 17c` to check out this version of the application. If you are using Microsoft Windows, run `git checkout 17c-waitress` to check out a version of the application configured to use the Waitress web server instead of Gunicorn.

Testing with Heroku Local

The Heroku CLI includes the `local` command, used to run the application locally in a very similar way to how it runs on the Heroku servers. However, environment variables such as `FLASK_APP` are not available when running the application locally. The `heroku local` command looks for environment variables that configure the application in a file named *.env* in the top-level directory of the application. For example, the *.env* file can contain the following variables:

```
FLASK_APP=flasky.py
FLASK_CONFIG=heroku
MAIL_USERNAME=<your-gmail-username>
MAIL_PASSWORD=<your-gmail-password>
```

Because the *.env* file contains passwords and other sensitive account information, it should never be added to source control.

Before the application can be started, the deployment task needs to be executed to set up the database. One-off tasks can be executed with the `local:run` command:

```
(venv) $ heroku local:run flask deploy
[OKAY] Loaded ENV .env File as KEY=VALUE Format
INFO  Context impl SQLiteImpl.
INFO  Will assume non-transactional DDL.
INFO  Running upgrade  -> 38c4e85512a9, initial migration
INFO  Running upgrade 38c4e85512a9 -> 456a945560f6, login support
INFO  Running upgrade 456a945560f6 -> 190163627111, account confirmation
INFO  Running upgrade 190163627111 -> 56ed7d33de8d, user roles
INFO  Running upgrade 56ed7d33de8d -> d66f086b258, user information
INFO  Running upgrade d66f086b258 -> 198b0eebcf9, caching of avatar hashes
INFO  Running upgrade 198b0eebcf9 -> 1b966e7f4b9e, post model
INFO  Running upgrade 1b966e7f4b9e -> 288cd3dc5a8, rich text posts
INFO  Running upgrade 288cd3dc5a8 -> 2356a38169ea, followers
INFO  Running upgrade 2356a38169ea -> 51f5ccfba190, comments
```

The `heroku local` command reads the *Procfile* and executes the tasks defined by it:

```
(venv) $ heroku local
[OKAY] Loaded ENV .env File as KEY=VALUE Format
11:37:49 AM web.1 |  [INFO] Starting gunicorn 19.7.1
11:37:49 AM web.1 |  [INFO] Listening at: http://0.0.0.0:5000 (91686)
11:37:49 AM web.1 |  [INFO] Using worker: sync
11:37:49 AM web.1 |  [INFO] Booting worker with pid: 91689
```

The logging output of all the tasks started by this command is consolidated into a single stream that is printed to the console, with each line prefixed with a timestamp and the task name.

The `heroku local` command also allows simulation of the use of multiple dynos to scale the application. The following command starts three web workers, each listening on a different port:

```
(venv) $ heroku local web=3
```

Deploying with git push

The final step in the process is to upload the application to the Heroku servers. Make sure that all the changes are committed to the local Git repository and then use `git push heroku master` to upload the application to the `heroku` remote:

```
$ git push heroku master
Counting objects: 502, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (426/426), done.
Writing objects: 100% (502/502), 108.03 KiB | 0 bytes/s, done.
Total 502 (delta 303), reused 146 (delta 61)
remote: Compressing source files... done.
remote: Building source:
remote:
remote: -----> Python app detected
remote: -----> Installing python-3.6.2
remote: -----> Installing pip
remote: -----> Installing requirements with pip
...
remote: -----> Discovering process types
remote:        Procfile declares types -> web
remote:
remote: -----> Compressing...
remote:        Done: 49.4M
remote: -----> Launching...
remote:        Released v8
remote:        https://<appname>.herokuapp.com/ deployed to Heroku
remote:
remote: Verifying deploy... done.
To https://git.heroku.com/<appname>.git
 * [new branch]      master -> master
```

The application is now deployed and running, but it is not going to work correctly because the `deploy` command that initializes the database tables has not been executed yet. The Heroku client can run this command as follows:

```
$ heroku run flask deploy
Running flask deploy on <appname>... up, run.3771 (Free)
INFO  [alembic.runtime.migration] Context impl PostgresqlImpl.
INFO  [alembic.runtime.migration] Will assume transactional DDL.
...
```

After the database tables are created and configured, the application can be restarted so that it starts cleanly with an updated database:

```
$ heroku restart
Restarting dynos on <appname>... done
```

The application should now be fully deployed and online at *https://<appname>.herokuapp.com*.

Reviewing application logs

The logging output generated by the application is captured by Heroku. To view the contents of the log, use the `logs` command:

```
$ heroku logs
```

During testing it can also be convenient to "tail" the log file, which can be done as follows:

```
$ heroku logs -t
```

Deploying an Upgrade

When a Heroku application needs to be upgraded the same process needs to be repeated. After all the changes have been committed to the Git repository, the following commands perform an upgrade:

```
$ heroku maintenance:on
$ git push heroku master
$ heroku run flask deploy
$ heroku restart
$ heroku maintenance:off
```

The `maintenance` option available on the Heroku CLI will take the application offline during the upgrade and will show a static page that informs users that the site will be coming back soon. This prevents users from accessing the application while it is going through the upgrade process.

Docker Containers

You are now familiar with Heroku, which is a fairly high-level deployment option. In this section you will learn how to work with *containers*, and in particular with the Docker platform, which is not as automated as a PaaS but provides more flexibility and is not tied to a specific cloud provider.

Containers are a special type of virtual machine that run on top of the kernel of the host operating system, unlike standard virtual machines, which have their own virtualized kernel and hardware. Because the virtualization stops at the kernel, containers are much more lightweight and efficient than virtual machines, but they require dedicated support built into the operating system. The Linux kernel has full support for containers.

Installing Docker

The most popular container platform is Docker, which has a free Community Edition (known as *Docker CE*) and a subscription-based Enterprise Edition (*Docker EE*). Docker can be installed on the three major desktop operating systems, and also on cloud servers. The easiest way to develop and test a "containerized" application is to install Docker CE on your development system. For macOS and Microsoft Windows there are one-click installers available from the Docker Store. This page also includes installation instructions for CentOS, Fedora, Debian, and Ubuntu Linux distributions.

After you complete the installation of Docker CE on your system, you should be able to access the `docker` command from your terminal:

```
$ docker version
Client:
 Version:      17.06.0-ce
 API version:  1.30
 Go version:   go1.8.3
 Git commit:   02c1d87
 Built:        Fri Jun 23 21:31:53 2017
 OS/Arch:      darwin/amd64

Server:
 Version:      17.06.0-ce
 API version:  1.30 (minimum version 1.12)
 Go version:   go1.8.3
 Git commit:   02c1d87
 Built:        Fri Jun 23 21:51:55 2017
 OS/Arch:      linux/amd64
 Experimental: true
```

Docker for Windows requires Microsoft's Hyper-V feature to be enabled. The installer will normally enable it for you, but if Docker does not appear to work correctly after installation, the state of the Hyper-V hypervisor is the first thing to check. You should keep in mind that enabling Hyper-V on your Windows machine will prevent other hypervisors (such as Oracle's VirtualBox) from working. If your system does not support Hyper-V virtualization, or you need a Docker solution that does not render other virtualization technologies unusable, you may want to install Docker Toolbox, a legacy Docker product for Windows that is based on VirtualBox.

Building a Container Image

The first task when working with containers is to build a container image for the application. An image is a snapshot of a container's filesystem, used as a template when starting new containers. Docker expects the instructions to create the image to be provided in a file named *Dockerfile*. Example 17-9 shows a *Dockerfile* that builds the application featured in this book.

Example 17-9. Dockerfile: container image build script

```
FROM python:3.6-alpine

ENV FLASK_APP flasky.py
ENV FLASK_CONFIG docker

RUN adduser -D flasky
USER flasky

WORKDIR /home/flasky

COPY requirements requirements
RUN python -m venv venv
RUN venv/bin/pip install -r requirements/docker.txt

COPY app app
COPY migrations migrations
COPY flasky.py config.py boot.sh ./

# runtime configuration
EXPOSE 5000
ENTRYPOINT ["./boot.sh"]
```

The build commands that can be included in a *Dockerfile* are documented in detail in the *Dockerfile* reference. In essence, these are deployment commands that install and configure the application in the container's filesystem, which is isolated from your system.

The FROM command is required in all *Dockerfiles* to specify a base container image to start from. In most cases, this is going to be an image that is publicly available in Docker Hub, Docker's container image repository. The repository contains official images for several Python interpreter versions. These are images that have a base operating system with Python installed on it. Images are specified with a name and a tag. The name of the official Docker Hub Python image is simply python. The different tags that are available can be seen in the Docker Hub page for the image. For the python image, tags are used to specify the desired interpreter version and platform. For this application, a 3.6 interpreter built on top of the Alpine Linux distribution is used. Alpine Linux is a platform commonly used in container images due to its small size.

> The macOS and Windows versions of Docker are able to run Linux-based containers.

The ENV command defines runtime environment variables. This command takes two arguments: a variable name and its value. Any environment variables defined with this command will be available when a container based on this image is executed. The FLASK_APP variable required by the flask command is defined here, as is FLASK_CONFIG, which is the name of the configuration class the application uses to configure itself when it starts. The Docker deployment will use a new configuration called docker, implemented in a DockerConfig class as shown in Example 17-10. This new configuration class inherits from ProductionConfig and just configures logging to be directed to stderr, which Docker automatically captures and exposes through the docker logs command.

Example 17-10. config.py: Docker configuration

```
class DockerConfig(ProductionConfig):
    @classmethod
    def init_app(cls, app):
        ProductionConfig.init_app(app)

        # log to stderr
        import logging
        from logging import StreamHandler
        file_handler = StreamHandler()
        file_handler.setLevel(logging.INFO)
        app.logger.addHandler(file_handler)

config = {
    # ...
```

```
    'docker': DockerConfig,
    # ...
}
```

The RUN command executes a command in the context of the container image. In the first occurrence of RUN, a *flasky* user is created inside the container. The adduser command is part of Alpine Linux, and is available in the base image selected by the FROM command. The -D argument to adduser suppresses an interactive prompt for the user's password.

The USER command selects the user under which the container will run, and also the user for the remaining commands in the *Dockerfile*. Docker uses the *root* user by default, but it is considered a good practice to switch to a regular user when root access isn't needed.

The WORKDIR command defines the top-level directory where the application is going to be installed. For this application, the home directory for the newly created *flasky* user is used. The remaining commands in the *Dockerfile* will execute with this directory as the current directory.

The COPY command copies files from the local filesystem to the container's filesystem. The *requirements*, *app*, and *migrations* directories are copied in their entirety, and then the top-level *flasky.py*, *config.py*, and new *boot.sh* files (discussed shortly) are copied as well.

The two additional RUN commands create a virtual environment and install the requirements in it. A dedicated requirements file was created for Docker as *requirements/docker.txt*. This file imports all the dependencies from *requirements/common.txt* and adds Gunicorn, which will be used as a web server as in the Heroku deployment.

The EXPOSE command defines the port on which the application running inside the container will install its server. When the container is started, Docker will map this port to a real port on the host machine, so that the container can receive requests from the outside world.

The final command is ENTRYPOINT. This command specifies how to execute the application when the container is started. The new *boot.sh* file, copied into the container above, is used as the startup script. Example 17-11 shows the contents of this file.

Example 17-11. boot.sh: container startup script

```
#!/bin/sh
source venv/bin/activate
flask deploy
exec gunicorn -b 0.0.0.0:5000 --access-logfile - --error-logfile - flasky:app
```

The script starts by activating the `venv` virtual environment that was created as part of the build. Then it runs the application's `deploy` command, built earlier in this chapter and also used for the Heroku deployment. This will create a new database, upgrade it to the latest version, and insert the default roles. Because the `DATABASE_URL` environment variable hasn't been set, the database will use the SQLite engine. Then a Gunicorn server listening on port 5000 is started. Docker captures all the output from the application and presents it as logs, so Gunicorn is configured to write both its access and error log files to standard output. Starting Gunicorn with `exec` makes the Gunicorn process take over the process running the *boot.sh* file. This is done because Docker pays special attention to the process that starts a container, and expects it to be the main process throughout its life. When this process ends, the container ends as well.

> If you have cloned the application's Git repository on GitHub, you can run `git checkout 17d` to check out this version of the application.

A container image for Flasky can now be built as follows:

```
$ docker build -t flasky:latest .
Sending build context to Docker daemon  51.08MB
Step 1/14 : FROM python:3.6-alpine
 ---> a6beab4fa70b
...
Successfully built 930e17a89b42
Successfully tagged flasky:latest
```

The `-t` argument to `docker build` gives a name and tag to the container image, separated by a colon. The `latest` tag name is typically used for the most up-to-date version of a container image. The dot at the end of the `build` command sets the current directory as the top-level directory during the build. Docker will look for the *Dockerfile* in this directory, and will also make the files in this directory and all subdirectories available to be added to the container image.

As a result of a successful `docker build` command, the built container image is stored in a local image repository. The `docker images` command shows the contents of the image repository on your system:

```
$ docker images
REPOSITORY    TAG          IMAGE ID      CREATED         SIZE
flasky        latest       930e17a89b42  5 minutes ago   127MB
python        3.6-alpine   a6beab4fa70b  3 weeks ago     88.7MB
```

This listing includes the just-built `flasky:latest` image and also the base Python 3.6 interpreter image referenced in the `FROM` *Dockerfile*, which Docker downloads and installs as part of the build.

Running a Container

Once a container image for the application is built, all that remains is to run it. The `docker run` command makes this a very simple task:

```
$ docker run --name flasky -d -p 8000:5000 \
  -e SECRET_KEY=57d40f677aff4d8d96df97223c74d217 \
  -e MAIL_USERNAME=<your-gmail-username> \
  -e MAIL_PASSWORD=<your-gmail-password> flasky:latest
```

The `--name` option gives the container a name. Naming containers is optional; if a name is not given, Docker generates one using randomly selected words.

The `-d` option starts the container in *detached* mode, which means that the container will run in the background on your system. A container that is not detached runs as a foreground task attached to the console session.

The `-p` option maps port 8000 in the host system to port 5000 inside the container. Docker provides the flexibility of mapping container ports to any port in the host system. This mapping enables two or more instances of the same container image to run on different host ports, while each instance uses its own virtualized port 5000.

The `-e` option defines environment variables that are going to exist in the context of the container, in addition to any variables defined at build time with the `ENV` command in the *Dockerfile*. The value assigned to the `SECRET_KEY` variable ensures that user sessions and tokens are signed with a unique and very hard to guess key. You should generate your own unique key for this variable. The values for the `MAIL_USERNAME` and `MAIL_PASSWORD` variables configure email sending through the Gmail service. For a production deployment that uses a different email service provider the `MAIL_SERVER`, `MAIL_PORT`, and `MAIL_USE_TLS` variables should be defined as well.

The final argument in the `docker run` command is the container image and tag to execute. This should match the name and tag given as the `-t` option to the `docker build` command.

When the container starts in the background, the `docker run` command prints the container ID to the console. This is a 256-bit unique identifier printed in hexadecimal notation. This ID can be used in any commands that require a reference to a container (in practice, only the first few characters of the ID need to be provided, such that the container can be uniquely identified).

To confirm that the container is running, the `docker ps` command can be used:

```
$ docker ps
CONTAINER ID  IMAGE          CREATED     STATUS     PORTS                    NAMES
71357ee776ae  flasky:latest  4 secs ago  Up 8 secs  0.0.0.0:8000->5000/tcp flasky
```

Since the container is now up and running, you can access the containerized application on port 8000 of your system, either locally as *http://localhost:8000* or from any other computer in the network as *http://<ip-address>:8000*.

To stop this container, use the `docker stop` command:

```
$ docker stop 71357ee776ae
71357ee776ae
```

The `stop` command stops the container but does not remove it from the system. To remove it, use the `docker rm` command:

```
$ docker rm 71357ee776ae
71357ee776ae
```

These two operations can be combined into one with `docker rm -f`:

```
$ docker rm -f 71357ee776ae
71357ee776ae
```

Inspecting a Running Container

When a container appears to misbehave, it might be necessary to debug it. The most obvious debugging mechanism is to add logging statements to the application and then monitor the running container with the `docker logs` command.

In some situations, however, it might be more convenient to open a shell session on the running container so that it can be inspected more closely. The `docker exec` command makes this possible:

```
$ docker exec -it 71357ee776ae sh
```

In this example, Docker is going to open a shell session with `sh` (the Unix shell) without interrupting the container. The `-it` options connect the terminal session from which the command is issued to the new process, so that the shell can be operated interactively. If the container includes other, more advanced shells such as *bash* or even a Python interpreter, they can be used as well.

A common strategy when troubleshooting containers is to create a special image loaded with additional tools such as a debugger that can later be invoked from a shell session.

Pushing Your Container Image to an External Registry

Having a container image locally is convenient when developing and testing an application, but when you are ready to share the image with others, you have to *push* it to an external registry server.

The Docker Hub registry is Docker's image repository, a convenient service where you can host your images. A free Docker Hub account allows you to store an unlimited number of public container images, but only one private image. Paid plans increase the number of private images you can host. To create your Docker Hub account, go to *https://hub.docker.com*.

Once you have a Docker Hub account, you can log in to it from the command line with the `docker login` command:

```
$ docker login
Login with your Docker ID to push and pull images from Docker Hub.
Username: <your-dockerhub-username>
Password: <your-dockerhub-password>
Login Succeeded
```

> To log in to a container image repository other than Docker Hub, pass the address of your repository as an argument to `docker login`.

Local container images are given a simple name. To prepare to push an image to Docker Hub, the image name must be prefixed with the Docker Hub account name and a slash as a separator. The `flasky:latest` image built earlier can be given a secondary name properly formatted for pushing to Docker Hub with the `docker tag` command:

```
$ docker tag flasky:latest <your-dockerhub-username>/flasky:latest
```

To upload the image to Docker Hub, use the `docker push` command:

```
$ docker push <your-dockerhub-username>/flasky:latest
```

The container image is now publicly available, and anybody can start a container based on it with the `docker run` command:

```
$ docker run --name flasky -d -p 8000:5000 \
<your-dockerhub-username>/flasky:latest
```

Using an External Database

One disadvantage of the way Flasky was deployed as a Docker container is that the default SQLite database lives in the same container as the application. This makes it very difficult to perform an upgrade, because once a running container is stopped, the database is gone with it.

A better approach is to host the database server separately from the application container. That makes upgrading the application while preserving the database an easy task, since all that is needed is to replace the application container with a new one.

Docker promotes a modular approach to building an application, in which each service is hosted in its own container. There are public container images available for MySQL, Postgres, and many other database servers. The `docker run` command can be used to deploy any of these directly to your system. The following command deploys a MySQL 5.7 database server to your system:

```
$ docker run --name mysql -d -e MYSQL_RANDOM_ROOT_PASSWORD=yes \
  -e MYSQL_DATABASE=flasky -e MYSQL_USER=flasky \
  -e MYSQL_PASSWORD=<database-password> \
  mysql/mysql-server:5.7
```

This command creates a container named `mysql` that runs in the background. The `-e` option assigns a few environment variables that this container takes as configuration. These and many other variables are documented in the Docker Hub page for the MySQL image. The preceding command configures the database with a randomly generated root password (use `docker logs mysql` right after starting the container to see the assigned password in the logs), and with a brand-new database called `flasky` that is configured to be accessed by a user named *flasky* as well. You need to provide a secure password for this user as a value for the `MYSQL_PASSWORD` environment variable.

To be able to connect to a MySQL database, SQLAlchemy requires a supported MySQL client package such as `pymysql` to be installed. This package can be added to the *docker.txt* requirements file.

> If you have cloned the application's Git repository on GitHub, you can run `git checkout 17e` to check out this version of the application.

The change made to the *requirements/docker.txt* file requires the container image to be rebuilt:

```
$ docker build -t flasky:latest .
```

If you are still running the previous application container, stop it and remove it with `docker rm -f`. Then start a new container with the updated application:

```
$ docker run -d -p 8000:5000 --link mysql:dbserver \
  -e DATABASE_URL=mysql+pymysql://flasky:<database-password>@dbserver/flasky \
  -e MAIL_USERNAME=<your-gmail-username> -e MAIL_PASSWORD=<your-gmail-password> \
  flasky:latest
```

There are two additions to the `docker run` command shown here. The `--link` option configures a connection between the new container and another existing one. The argument to `--link` consists of two names separated by a colon: the source container name or ID, and an alias for that container in the container being created. In this example the source container is `mysql`, the database container started earlier. This container is going to be accessible in the new Flasky container with the `dbserver` hostname.

To complete the configuration, a `DATABASE_URL` environment variable is added, with a connection URL that points to the `flasky` database in the `mysql` container. The `dbserver` alias is used as the database host, as Docker makes sure that this name resolves to the IP address of the linked container. The value of the `MYSQL_PASSWORD` environment variable set in the `mysql` container must be included in the connection URL for this container as well. The value of `DATABASE_URL` overrides the default SQLite database, so with this simple change the container will be configured to connect to the MySQL database.

> The Docker Hub repository is a gold mine of very useful applications and services that are packaged and ready to use in a Docker environment, either standalone or as base images for your own containers. You will find that all sorts of projects (including databases, web servers, load balancers, programming languages, operating systems, and more) offer official images.

Container Orchestration with Docker Compose

Containerized applications are usually composed of several running containers. You have seen in the previous section that the main application and the database server run in independent containers. As the application grows in complexity, it will invariably need more containers. Some applications are going to require additional services, such as message queues or caches. Other applications may take advantage of a *microservices* architecture and have a distributed structure with several smaller subapplications, each running in its own container. Applications that have to handle high loads or need to be fault-tolerant will want to *scale out* by running several instances behind a load balancer.

As the number of containers that are part of the application increases, the task of managing and coordinating all these containers is going to become much harder if Docker alone is used. Container *orchestration* frameworks built on top of Docker help with this task.

The *Compose* toolset provides basic orchestration functionality, included with the Docker installation. With Compose, the containers that are part of an application are described in a configuration file, typically named *docker-compose.yml*. The `docker-compose` command can then start all the containers associated with the application using a single command.

Example 17-12 shows a *docker-compose.yml* file that represents the containerized Flasky along with its MySQL service.

Example 17-12. docker-compose.yml: compose configuration

```
version: '3'
services:
  flasky:
    build: .
    ports:
      - "8000:5000"
    env_file: .env
    links:
      - mysql:dbserver
    restart: always
  mysql:
    image: "mysql/mysql-server:5.7"
    env_file: .env-mysql
    restart: always
```

This file is written in YAML, which is a clean and simple format that can represent hierarchical structures that are composed of key-value maps and lists. The `version` key specifies which version of Compose is used, and the `services` key defines the containers of the application as its children. In the case of Flasky, these are two services named `flasky` and `mysql`.

For services such as `flasky`, which are built as part of the application, the subkeys specify the arguments that are given to the `docker build` and `docker run` commands. The `build` key specifies the build directory, where the *Dockerfile* is located. The `ports` key specifies the network port mappings. The `env_file` key is a convenient way to define several environment variables that the container needs. The `links` key establishes a link to the MySQL container, by exposing it with the hostname `dbserver`. The `restart` key set to `always` provides a simple way for Docker to automatically restart the container if it exits unexpectedly. The *.env* file for this deployment should have the following variables in it:

```
FLASK_APP=flasky.py
FLASK_CONFIG=docker
SECRET_KEY=3128b4588e7f4305b5501025c13ceca5
MAIL_USERNAME=<your-gmail-username>
MAIL_PASSWORD=<your-gmail-password>
DATABASE_URL=mysql+pymysql://flasky:<database-password>@dbserver/flasky
```

The `mysql` service has a simpler structure, because this is a service that is started from a stock image that does not require a build step. The `image` key specifies the name and tag of the container image to use for this service. As with the `docker run` command, Docker will download this image from the container image registry. The `env_file` and `restart` keys are similar to those used in the `flasky` container. Note how the environment variables for the MySQL container are stored in a separate file named *.env-mysql*. While it would be easier to add the environment variables needed by all containers to the *.env* file, it is a good practice to prevent one container from having access to the secrets of another. The *.env-mysql* file needs the following environment variables defined:

```
MYSQL_RANDOM_ROOT_PASSWORD=yes
MYSQL_DATABASE=flasky
MYSQL_USER=flasky
MYSQL_PASSWORD=<database-password>
```

The *.env* and *.env-mysql* files contain passwords and other sensitive information, so they should never be added to source control.

A complete reference for the *docker-compose.yml* file is found at "the Docker website".

A typical problem with orchestrated systems is that containers are started in the wrong order—or in the correct order, but without giving the containers for base services enough time to start and initialize before starting higher-level containers that depend on them. In the case of Flasky, the `mysql` container needs to start first, so that the database is up and running when the `flasky` container starts. Then it can connect to the database, apply the database migrations, and finally start the web server.

Compose will start the `mysql` and `flasky` containers in the right order, because it will detect the dependency between them from the `links` key in the `flasky` container. But Compose is not going to wait for MySQL to start, which might take a few seconds. When designing distributed systems, it is a good practice to implement retries in all connections to external services. Example 17-13 shows how the *boot.sh* script that

starts the `flasky` container can be made more robust by retrying the `flask deploy` command, which retries the database upgrade until it succeeds.

Example 17-13. boot.sh: waiting for the database to be up

```
#!/bin/sh
source venv/bin/activate

while true; do
    flask deploy
    if [[ "$?" == "0" ]]; then
        break
    fi
    echo Deploy command failed, retrying in 5 secs...
    sleep 5
done

exec gunicorn -b :5000 --access-logfile - --error-logfile - flasky:app
```

By running `flask deploy` inside a retry loop, the container will be able to tolerate failures due to the database service not being immediately ready to accept requests.

> If you have cloned the application's Git repository on GitHub, you can run `git checkout 17f` to check out this version of the application. Also make sure that the *.env* and *.env-mysql* environment files are created and populated with values correct for your environment.

Now that the Compose configuration is complete, the application can be started with the `docker-compose up` command:

```
$ docker-compose up -d --build
```

The `--build` option to `docker-compose up` indicates that a build step should run before launching the application. This will cause the `flasky` container image to be built. After the image is created, the `mysql` and `flasky` containers will be started in that order. The `-d` option starts the containers in detached mode, as with the single container. After a few seconds, the application should be up and running in the background, and you should be able to connect to it at *http://localhost:8000*.

Compose consolidates the logging from all the containers into a single stream, which you can see with the `docker-compose logs` command:

```
$ docker-compose logs
```

Or, if you want to constantly monitor the log stream:

```
$ docker-compose logs -f
```

The `docker-compose ps` command shows a summary of all the application containers that are running and their state:

```
$ docker-compose ps
     Name                Command           State           Ports
-------------------------------------------------------------------------
flasky_flasky_1   ./boot.sh                Up      0.0.0.0:8000->5000/tcp
flasky_mysql_1    /entrypoint.sh mysqld    Up      3306/tcp, 33060/tcp
```

To upgrade an application to a new version, simply make the necessary changes to it and repeat the `docker-compose up` command used previously to start it. Compose will rebuild the application container if anything changed, and then replace the older container with a fresh one.

To stop the application, use the `docker-compose down` command, or `docker-compose rm --stop --force` if you also want to remove the stopped containers.

Cleaning Up Old Containers and Images

As you work with containers, your system will invariably accumulate old containers or images that are not needed anymore. It is a good idea to routinely review and clean those up, so that they don't take up space on the system.

To see the list of containers in the system, use the following command:

```
$ docker ps -a
```

This will show containers that are running, and containers that were stopped but are still in the system. To delete any containers from this list, use the `docker rm -f` command and provide the names or IDs to remove:

```
$ docker rm -f <name-or-id> <name-or-id> ...
```

To see the list of container images stored in your system, use the `docker images` command. If there are any images that you want to remove, you can do so with the `docker rmi` command.

Some containers create virtual *volumes* on the host computer that are used for storage outside of the container filesystem. The MySQL container image, for example, puts all the database files in a volume. You can view a list of all the allocated volumes in your system with `docker volume ls`. To remove a volume that is unused, use `docker volume rm`.

If you prefer a more automatic cleanup, the `docker system prune --volumes` command will remove any unused images or volumes, and any stopped containers that are still in the system.

Using Docker in Production

Many people consider Docker a development and testing platform only. While the techniques presented in the previous sections can be used to deploy applications on production servers running Docker, there are some limitations and security concerns that need to be considered:

Monitoring and alerting
: What happens if a containerized application crashes? Docker can restart a container that exits unexpectedly, but it will not monitor your containers, nor will it send alerts when they behave erratically.

Logging
: Docker maintains a separate log stream for each container. Compose improves this by offering a consolidated stream, but without long-term storage or searching and filtering capabilities.

Management of secrets
: Configuring passwords and other credentials through environment variables is insecure, since Docker exposes pre-configured environment variables via the `docker inspect` command or through its API.

Reliability and scaling
: To help with fault tolerance, or to accommodate increasing load demands, it is necessary to run several instances of the application on several hosts and behind one or more load balancers.

These limitations are generally addressed by more elaborated orchestration frameworks built on top of Docker or other container runtimes. Frameworks such as *Docker Swarm* (now incorporated into Docker), *Apache Mesos*, and *Kubernetes* are good choices for building robust container deployments.

Traditional Deployments

So far you have seen how Heroku and Docker manage deployments. To complete this review of deployment strategies, this section will describe a traditional hosting option, which involves buying or renting a server, either physical or virtual, and manually setting up all the required components on it. This is obviously the most laborious option of all, but it can be a convenient option when you have terminal access to production server hardware. The following sections will give you an idea of the work involved.

Server Setup

There are several administration tasks that must be performed on the server before it can host applications:

- Install a database server such as *MySQL* or *Postgres*. Using an *SQLite* database is also possible but is not recommended for a production server due to its many limitations with regard to modification of existing database schemas.
- Install a Mail Transport Agent (MTA) such as *Sendmail* or *Postfix* to send email out to users. Using Gmail in a production application is not possible, as this service has very restrictive quotas and specifically prohibits commercial use in its terms of service.
- Install a production-ready web server such as *Gunicorn* or *uWSGI*.
- Install a process-monitoring utility such as *Supervisor*, that immediately restarts the web server if it crashes or after the host is power-cycled.
- Install and configure an SSL certificate to enable secure HTTP.
- (Optional but highly recommended) Install a front-end reverse proxy web server such as *nginx* or *Apache*. This server is configured to serve static files directly and forward application requests into the application's web server, which is listening on a private port on *localhost*.
- Secure the server. This includes several tasks that have the goal of reducing vulnerabilities in the server such as installing firewalls, removing unused software and services, and so on.

> Instead of manually performing these tasks, create a scripted deployment using an automation framework such as Ansible, Chef, or Puppet.

Importing Environment Variables

Similarly to Heroku and Docker, an application running on a standalone server relies on certain settings such as the database connection URL, email server credentials, etc. being provided in environment variables.

Because there is no Heroku or Docker to configure these variables before the application starts, the procedure to set the variables is dependent on the platform and tools used. To make the configuration of environment variables easier and uniform across deployment platforms, the short code block in Example 17-14 imports into the environment a *.env* file similar to the one used with the `heroku local` and `docker-`

compose commands, using a Python package called python-dotenv that needs to be installed with *pip*. This is done in *flasky.py* before the application instance is created, so that by the time the configuration is imported these variables are accessible in the environment.

Example 17-14. flasky.py: importing the environment from the .env file

```
import os
from dotenv import load_dotenv

dotenv_path = os.path.join(os.path.dirname(__file__), '.env')
if os.path.exists(dotenv_path):
    load_dotenv(dotenv_path)
```

The *.env* file can define the FLASK_CONFIG variable that selects the configuration to use, the DATABASE_URL connection, the email server credentials, etc. As explained before, a *.env* file should not be added to source control due to the sensitive nature of some of the items in it.

> If you created a *.env* file for use with Heroku or Docker, review it and adjust it appropriately, because with the changes just made, the application will import the variables defined in this file for all configurations.

Setting Up Logging

For Unix-based servers, logging can be sent to the *syslog* daemon. A new configuration specifically for Unix can be created as a subclass of ProductionConfig, as shown in Example 17-15.

Example 17-15. config.py: Unix example configuration

```
class UnixConfig(ProductionConfig):
    @classmethod
    def init_app(cls, app):
        ProductionConfig.init_app(app)

        # log to syslog
        import logging
        from logging.handlers import SysLogHandler
        syslog_handler = SysLogHandler()
        syslog_handler.setLevel(logging.WARNING)
        app.logger.addHandler(syslog_handler)
```

With this configuration, application logs will be written to the configured syslog messages file, typically */var/log/messages* or */var/log/syslog* depending on the Linux distri-

bution. The syslog service can be configured to write a separate log file for application logs, or to send the logs to a different machine if desired.

> If you have cloned the application's Git repository on GitHub, you can run `git checkout 17g` to check out this version of the application.

CHAPTER 18
Additional Resources

You are pretty much done with this book. Congratulations! I hope the topics that I have covered have given you a solid base to begin building your own applications with Flask. The code examples are open source and have a permissive license, so you are welcome to use as much of my code as you want to seed your projects, even if they are of a commercial nature. In this short final chapter, I want to give you a list of additional tips and resources that might be useful as you continue working with Flask.

Using an Integrated Development Environment (IDE)

Developing Flask applications in an integrated development environment (IDE) can be very convenient, since features such as code completion and an interactive debugger can speed up the coding process considerably. Some of the IDEs that work well with Flask are listed here:

PyCharm
: IDE from JetBrains with Community (free) and Professional (paid) editions, both compatible with Flask applications. Available on Linux, macOS, and Windows.

Visual Studio Code
: Open source IDE from Microsoft. A third-party Python plug-in must be installed to have access to code completion and debugging features with Flask applications. Available on Linux, macOS, and Windows.

PyDev
: Open source IDE based on Eclipse. Available on Linux, macOS, and Windows.

Finding Flask Extensions

The examples in this book rely on several extensions and packages, but there are many more that are also useful and were not discussed. The following is a short list of some additional packages that are worth exploring:

- Flask-Babel: Internationalization and localization support
- Marshmallow: Serialization and deserialization of Python objects, useful for API resource representations
- Celery: Task queue for processing background jobs
- Frozen-Flask: Conversion of a Flask application to a static website
- Flask-DebugToolbar: In-browser debugging tools
- Flask-Assets: Merging, minifying, and compiling of CSS and JavaScript assets
- Flask-Session: Alternative implementation of user sessions that use server-side storage
- Flask-SocketIO: Socket.IO server implementation with support for WebSocket and long-polling

If the functionality that you need for your project is not covered by any of the extensions and packages mentioned in this book, then your first destination to look for additional extensions should be the official Flask Extension Registry. Other good places to search are the Python Package Index, GitHub, and Bitbucket.

Getting Help

If you reach a point where you are blocked by an issue that you cannot resolve on your own, you should keep in mind that there is a community of Flask developers like you that will be happy to help you.

A great place to ask questions about Flask or any related extensions is Stack Overflow. Other developers that see your question and know how to answer will post their answers, which are voted up or down according to their quality. You, as the owner of the question, can then select the best answer. All questions and their answers stay on the site and appear in search results. So, by asking your question on this platform, you help grow the collection of information about Flask.

Reddit also has a friendly Flask-dedicated subreddit where you can post questions.

Finally, if you use IRC, the `#pocoo` channel on Freenode is frequented by Flask developers of all levels who can help you one-on-one with your problem.

Getting Involved with Flask

Flask would not be as awesome as it is without the work done by its community of developers. As you are now becoming part of this community and benefiting from the work of so many volunteers, you should consider finding a way to give something back. Here are some ideas to help you get started:

- Review the documentation for Flask or your favorite related project and submit corrections or improvements.
- Translate the documentation into a new language.
- Answer questions on Q&A sites such as Stack Overflow.
- Talk about your work with your peers at user group meetings or conferences.
- Contribute bug fixes or improvements to packages that you use.
- Write new Flask extensions and release them as open source.
- Release your applications as open source.

I hope you decide to volunteer in one of these ways, or any others that are meaningful to you. If you do, thank you!

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

About the Author

Miguel Grinberg has over 25 years of experience as a software engineer. He has a blog where he writes about a variety of topics, including web development, robotics, photography, and the occasional movie review. He lives in Portland, Oregon.

Colophon

The animal on the cover of *Flask Web Development* is a Pyrenean Mastiff (a breed of *Canis lupus familiaris*). These giant Spanish dogs are descended from an ancient livestock guardian dog called the Molossus, which was bred by the Greeks and Romans and is now extinct. However, this ancestor is known to have played a role in the creation of many breeds that are common today, such as the Rottweiler, Great Dane, Newfoundland, and Cane Corso. Pyrenean Mastiffs have only been recognized as a pure breed since 1977, and the Pyrenean Mastiff Club of America is working to promote these dogs as pets in the United States.

After the Spanish Civil War, the population of Pyrenean Mastiffs in their native homeland plummeted, and the breed only survived due to the dedicated work of a few scattered breeders throughout the country. The modern gene pool for Pyreneans stems from this postwar population, making them prone to genetic diseases like hip dysplasia. Today, responsible owners make sure their dogs are tested for diseases and x-rayed to look for hip abnormalities before being bred.

Adult male Pyrenean Mastiffs can reach upwards of 200 pounds when fully grown, so owning this dog requires a commitment to good training and plenty of outdoor time. Despite their size and history as hunters of bears and wolves, the Pyrenean has a very calm temperament and is an excellent family dog. They can be relied upon to take care of children and protect the home, while at the same time being docile with other dogs. With proper socialization and strong leadership, the Pyrenean Mastiff thrives in a home environment and will provide an excellent guardian and companion.

Many of the animals on O'Reilly covers are endangered; all of them are important to the world. To learn more about how you can help, go to *animals.oreilly.com*.

The cover image is from J. G. Wood's *Animate Creation*. The cover fonts are URW Typewriter and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.